Timing Analysis of General-Purpose Graphics Processing Units for Real-Time Systems: Models and Analyses

---

A Dissertation Thesis by

Kostiantyn Berezovskyi

supervised by Dr. Konstantinos Bletsas

Submitted to the graduate faculty of the

Faculdade de Engenharia da Universidade do Porto

Departamento de Engenharia Electrotécnica e de Computadores

in partial fulfillment of the requirements

for the Dissertation Thesis and

subsequent Ph.D. in Electrical and Computer Engineering

Timing Analysis of General-Purpose Graphics Processing Units for Real-Time Systems: Models and Analyses

Kostiantyn Berezovskyi

Doctoral Program in Electrical and Computer Engineering

Approved by:

President: Dr. Aurélio Joaquim de Castro Campilho

External Referee: Dr. Shinpei Kato

External Referee: Dr. Christine Rochange

Internal Referee: Dr. Eduardo Manuel de Médicis Tovar

Internal Referee: Dr. José Alfredo Ribeiro da Silva Matos

Internal Referee: Dr. Pedro Alexandre Guimarães Lobo Ferreira Souto

Internal Referee: Dr. João Paulo de Castro Canas Ferreira

Supervisor: Dr. Konstantinos Bletsas

To my dear parents – Svitlana Petrivna and Anatolii Grygorovych.

Acknowledgements

The completion of this thesis was possible because of the supervision, knowledge, help and support from many beautiful people who I was lucky to meet.

Firstly, I would like to thank my supervisor. This dissertation simply would not have been possible without his contributions. Guiding me was an exhaustive workload that required much effort, planning and meetings on daily basis. I am grateful to Konstantinos Bletsas for his insightful ideas, fruitful suggestions, patience and optimism that he was demonstrating especially at those moments when the things were not running well on my side.

The role of my co-authors, without who I would not be able to complete none of those publications, was of paramount importance. I would like to thank Eduardo Tovar, Luca Santinelli, Fabrice Guet, Stefan M. Petters and Björn Andersson for their tireless work and selfless collaboration.

The influence of my professors would be hard to overestimate. I wish to thank Mário Sousa, Luís Almeida, José Fernando Oliveira, Luís Miguel Pinho, Mário Alves, Shashi Prabh, Sónia Amorim, Maria do Carmo Lopes, Jorge Porto, Helena Fernandes, Artur Carvalho and Anatolii Doroshenko for their fun classes and interesting assignments.

Addressing bureaucratic issues turned out to be a demanding task for me. I am grateful to Inês Almeida, Cristiana Barros and Sandra Almeida for their help.

I was lucky to get many useful advices from Pedro Souto, Luis Lino Ferreira, Filipe Pacheco, Michele Albano, Per Lindgren, Vincent Nlis, Geoffrey Nelissen, António Barros, Matthias Becker, Davide Compagnin, Arvind Easwaran and Andrea Baldovin.

Working with hardware was an interesting part of my work. I would like to thank André Ribeiro and Marwin Adorni for giving me a hand on that.

I had a chance to share office space with bright colleagues, who taught me a lot and also created competitive environment. Therefore, I would like to thank Artem

Burmyakov, Borislav Nikolic, Hazem Ali, José Marinho, Gurulingesh Raravi, Dakshina Dasari, Vikram Gupta, Suhas Aggarwal, Harrison Kurunathan, Shashank Gaur, André Pedro, Renato Ayres, Paulo Barbosa, Humberto Carvalho, Bruno Deveza, Joss Santos, José Bruno Silva and Muhammad Ali Awan.

My adaptation in Portugal would be much tougher without the help of my friends. I wish to express my gratitude to José Fonseca, Ricardo Severino, Paulo Baltarejo Sousa, João Loureiro, Alexandre Esper, Cludio Maia, David Pereira, José Augusto Santos and Pedro Santos for helping me to improve my Portuguese skills.

Getting in touch with the local culture was greatly facilitated thanks to the help of Tiago Martins, Krystallenia Batziou and Joana Ilhão.

Living abroad is not an easy experience, thus, I am thankful to Lorenzo Lanzi, Juliane Oliveira and Miguel Lanzi Oliveira for creating for me a family-environment far from home.

I would like to thank all the members of my dear family for their every-day support and for inspiring me with their scientific and engineering achievements.

I wish to thank wonderful Portugal and beautiful Porto for allowing me, the foreigner, to feel myself over here like being at home.

Finally, this work was partially supported by FCT under PhD grant SFRH/BD/82069/2011.

Declaration

Some parts of this thesis proposal have appeared in the previously published paper (the respective author is marked by the asterisk):

- K. Berezovskyi*, K. Bletsas, and B. Andersson. Makespan computation for GPU threads running on a single streaming multiprocessor. In Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS'2012), July 2012.

- K. Berezovskyi*, K. Bletsas, and S. M. Petters. Faster makespan estimation for GPU threads on a single streaming multiprocessor. In Proceedings of the 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2013), September 2013.

- K. Berezovskyi*, L. Santinelli, K. Bletsas, and E. Tovar. WCET Measurement-based and Extreme Value Theory Characterisation of CUDA Kernels. In Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS 2014), October 2014.

- K. Berezovskyi*, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar. Measurement-Based Probabilistic Timing Analysis for Graphics Processing Units. Accepted to the 29th International Conference on Architecture of Computing Systems (ARCS 2016).

Resumo

Originalmente, os GPUs (*Graphics Processing Units*) foram desenvolvidos especificamente para acelerar a renderização gráfica. Hoje em dia, esta tecnologia suporta o processamento das mais diversas operações computacionais, o que faz com que seja amplamente usada de forma a retirar carga ao CPU (*Central Processing Unit*) e libertar outros recursos do sistema. Em particular, os GPUs são adequados para solucionar problemas computacionais massivamente paralelos, uma vez que gerem com eficiência a interação entre uma grande quantidade de *threads* de curta duração e as unidades de processamento.

A utilização de GPUs, em sistemas embebidos, implica o desenvolvimento de análises que permitam calcular os limites no tempo de execução das GPU-threads, já que as análises existentes para CPUs não são aplicáveis. O que é fundamental neste caso, não é saber quanto tempo demora a computação de cada uma das GPU-threads, mas sim quanto tempo demora para que todas concluam a execução, tendo em consideração a competição que se verifica no acesso aos recursos do GPU.

Nesta dissertação, nós desenvolvemos abordagens teóricas e práticas para a análise temporal de tarefas paralelas a serem processadas por GPUs. Mais propriamente, o objectivo é fornecer valores exactos ou limites susperiores próximos do exacto, limites superiores probabilísticos, e limites inferiores marginalmente otimistas, em relação àquilo que é o pior comportamento temporal na sequência de execução das tarefas no GPU. Estas abordagens são designadas *optimization-based*, *probabilistic measurement-based* e *metaheuristic-based*, respectivamente. A sua formulação tem em conta as característica do *hardware*, restrições de tratabilidade e algumas suposições convenientes.

Abstract

Graphics processors were originally developed for rendering graphics but have evolved towards being an architecture for general-purpose computations. These processors are well-suited for massively parallel computational problems because of the ability to efficiently manage a great number of lightweight threads competing for the computational resources of the processor. Today, Graphics Processing Units (GPUs) are widely used to unload Central Processing Units (CPUs), liberate other resources of a given computer system, and provide an alternative to multiprocessor computers as a means of processing computationally expensive parallel tasks. The recent trend of utilizing GPUs in embedded systems necessitates developing timing analysis approaches for finding bounds on the execution time of GPU-threads because the approaches developed for CPU timing analysis are not applicable. The reason is that we are not interested in how long it takes for any given GPU thread to complete, but rather how long it takes for all of the GPU threads to complete in the context of their competition for the functional units.

We developed both theoretical and practical approaches for GPU timing analysis that could provide exact values and tight upper bounds, marginally optimistic lower bounds or probabilistic upper bounds on the worst-case temporal behavior of GPU processing. We call these approaches optimization-based, metaheuristic-based and statistical measurement-based respectively. We formulate them subject to the hardware features, tractability constraints and some simplifying assumptions.

First, we proposed a model of a single streaming multiprocessor – a computationally independent module of a GPU. The optimization-based and metaheuristic-based approaches are formulated in the context of that theoretical model and related assumptions. The measurement-based approach is targeting the real GPU hardware and is ready for practical usage.

The optimization-based approach is built upon a simple but very pessimistic tech-

nique for finding an upper bound on the worst-case makespan – the longest possible time interval between the moment when the "earliest" GPU thread starts its execution, and the moment when the "latest" thread finishes. The outcome of this technique is used for the formulation of a combinatorial optimization problem for finding an exact value of the worst-case execution requirement. Addressing the issue of tractability, we also proposed a marginally pessimistic estimation technique for finding a tight upper-bound on the worst-case makespan. This approach was implemented in a timing analysis software tool applicable to the problem instance under consideration subject to the configuration of the streaming multiprocessor.

Pursuing an objective of discovering computationally fast approaches we addressed the problem of finding the worst-case makespan from the metaheuristic viewpoint. We experimentally demonstrated that the metaheuristic-based approach is able to find a tight lower bound and in combination with the optimization-based approach proposes a complete framework for bounding the respective solution from both, the top and the bottom. This aspect is of paramount importance for the cases when an exact worst-case makespan of the problem under consideration cannot be tractably computed. On the other hand, the simplicity, flexibility and ability for massive parallelization of the metaheuristic-based approach determine a potential of its usage for soft real-time systems.

Aiming to bring our research closer to the industry, in order to overcome some limiting assumptions of memory subsystem, we addressed the problem of GPU timing analysis from the probabilistic and measurement-based perspectives. Our statistical measurement-based approach includes a marginally invasive technique for obtaining the GPU execution time measurements. For analyzing these measurements, the approach introduced a probabilistic characterization of the worst-case temporal behavior of GPU applications. We formulated our approach based on a solid statistical background of Extreme Value Theory (EVT) and the "Block Maxima" paradigm. The

applicability of EVT was extended to less constraining hypotheses than independence. We also provided a way for obtaining accurate estimates on the worst-case execution requirement for the desired confidence level.

Outline

List of Figures

List of Tables

1 Introduction

The massive computational power of Graphics Processing Units (GPUs), combined with novel programming models such as CUDA [156], makes them attractive platforms for many parallel applications. For example, for signal processing applications, a GPU is a good choice for a platform due to their availability and highly developed software ecosystem. This also includes embedded and real-time applications, which, however, also have temporal constraints: computations must not only be correct but also completed on time. This poses a challenge because the characterization of the worst-case temporal behavior of parallel applications on GPUs is still an open problem.

1.1 Problem Statement

To provide temporal guarantees for GPU-accelerated applications, we need approaches for upper-bounding their execution time on the GPU. Traditional Worst-Case Execution Time (WCET) [182] analyses for Central Processing Units (CPUs) are inapplicable because they focus on the WCET of a single entity of execution (i.e., a *thread*). Yet, on GPUs the result is pieced together from thousands of threads, competing for GPU resources, and we are not interested in the WCET of any single thread in particular. Rather, we seek to bound the time, from when the earliest GPU thread starts executing until all of them have completed. On the other hand, the evidence shows that the timing analysis techniques developed for CPUs cannot even be considered as applicable to graphics processors because of the crucial differences of CPU and GPU architectures. The need for GPU timing analysis, that real-time embedded system community faces these days, is reflected in the novel research topic of this dissertation: “Timing Analysis of General Purpose Graphics Processor Units for Real-Time Systems”.

1.2 Research Approach

Given that today's GPU architectures are subject to substantial changes in between revisions to the hardware, many of their important features (e.g., internal scheduling policy) are left undocumented. This gives to the chip-makers the freedom of taking different technological paths and making experimental designs, but on the other hand it poses a challenge for the researchers and engineers who make timing analysis for these hardware architectures and target meeting the timing requirements of the real-time systems powered by the GPUs. Our way of addressing this problem could be briefly described via the following two research directions:

- developing static approaches for finding upper and lower bounds on the kernel execution time;
- formulating and validating a Measurement-Based Probabilistic Timing Analysis approach (MBPTA) based on Extreme Value Theory (EVT).

The process of developing static approaches is subject to the following high-level steps:

- creating models of the GPU hardware by giving the preference to pessimistic rather than optimistic assumptions;
- developing and implementing the techniques for obtaining the bounds on the worst-case execution timings of GPU kernels subject to the models under consideration.

Unlike the approaches mentioned above, probabilistic measurement-based approaches target directly the hardware, rather than its models. A worst-case execution requirement estimate, provided by an approach of such kind, is subject to a probability that the respective estimate will not be exceeded. These approaches are based on the following two stages:

- creating techniques for profiling GPU kernel execution on real hardware with the least possible measurement overhead and collecting the corresponding timings;
- applying Extreme Value Theory (EVT) to the measurements collected during the previous stage for the sake of providing an accurate probabilistic worst-case estimate.

1.3 Thesis Statement

Elliott demonstrated [56] that the use of GPUs is beneficial for real-time systems and such an integration is expected to be effective in real-life scenarios. We use the statement of his dissertation as a basis for our motivation, which allows us to concentrate on addressing a timing analysis problem which is required for a successful application of GPUs in the real-time domain. Therefore, the statement of this thesis is the following:

> The problem of GPU timing analysis can be successfully addressed in the context of real-time systems. The resulting approaches represent the range of modern timing analysis research: from static to measurement-based, subject to the strictness of timeliness guarantees of the respective real-time application. These techniques have a potential to satisfy the future industrial needs.

1.4 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 introduces computing systems. Chapter 3 presents the literature review. Chapter 4 introduces the model of GPU chip architecture and the GPU programming model. Chapter 5 discusses an approach based on optimization problems. Chapter 6 introduces a metaheuristic-based approach. Chapter 7 presents a probabilistic approach based on measurements,

Chapter 8 discusses future work directions and concludes.

2 Background on computing systems

Let us consider a *signal* as a transmitted energy from which some information can be obtained. An *information* is that which *informs* and from which *data* can be derived. Data refers to some information that is *coded* or represented in some form being amenable for *processing* or usage. An *information processing system* is a system which takes information in one form and processes it into another form by deriving data and organizing it according to some *logic*. In this thesis we consider a *computing system* to be an electrical information processing system organized as a combination of two subsytems: *hardware* and *software*.

Hardware is the collection of physical components. It includes both, essential components that are necessary for the computing system to function, and auxiliary components that provide additional functionalities. All these components process instructions, where each instruction is an atomic operation supported by the hardware (subject to the respective functional requirements implemented).

Sequences of instructions, grouped together according to some logic, form software. It is the software that specifies the workload to be performed by the hardware. Therefore, hardware and software have to work together for the sake of forming a usable computing system.

Among computing systems there is a distinction between two broad categories: *embedded systems* and *general-purpose systems*.

2.1 Embedded and general-purpose computing systems

An embedded system is designed to be a subsystem of a more complex system that includes other electrical parts, mechanical parts, etc. Therefore, such a computing system is *embedded* as part of larger *host system*[1] [181]. Usually, embedded systems

[1]In this thesis, we use the term *host system* for a bigger system that includes the embedded system under consideration. Note, that this is a different meaning comparing to *end system* from

are characterized by a fixed set of dedicated tasks to be performed. In this sense, an embedded system is custom-made for a specific application domain subject to concerns regarding functional and non-functional requirements of the host system.

The concept of an embedded system is tightly related to the concept of a *controller* – a device that monitors and controls the operation of a given dynamical system, e.g., maintaining settings for liquid flow, temperature, pressure, etc. Historically, controllers were implemented by combining mechanical, pneumatic and hydraulic components. However, rapid development in electronic science and technology has brought a huge variety of electronic controllers. Although the term *controller* can be used to refer to a stand-alone controlling device, more often a controller is implemented as an electronic circuit assembled of electronic components connected by wires or traces that provide conductivity for electric current flows. In this case, the controller is in the "heart" of the embedded system, managing and interfacing with other its parts.

A general-purpose computing system, as opposed to an embedded one, is designed to be stand-alone. It has to be configurable and suitable for a broad range of workloads. The hardware of such systems has potential for augmented functionality while the software often needs to be frequently updated or even replaced. Unlike embedded systems, general-purpose systems do not usually have so strict requirements on power consumption, size and price per unit. Usually, they are not expected to be used in harsh operational conditions, therefore, in the average case, the level of reliability of a general-purpose system can be significantly lower. Because of all such aspects, designers of general-purpose systems often have more freedom in trying new approaches and experimenting with altering configurations. This is the reason of a rapid progress in general-purpose hardware and software which also leads to migration of many general-purpose features to embedded systems domain.

the networking domain of computer engineering, that is sometimes referred to as host system in networking jargon.

The "heart" of general-purpose hardware is the *processor* – a component able to carry out a set of supported arithmetic, logical or control operations. Unlike a controller in embedded systems, a processor has to be suitable for a much broader range of operations, hence, its design is often more complex.

Both embedded and general-purpose computing systems development is greatly influence by two competing hardware architectures: *Princeton* design and *Harvard* design.

2.2 Princeton and Harvard architectures

The Princeton hardware architecture [54] also known as the *von Neumann computer* [178] was dominant in the early years of computer engineering. The basic scheme of that architecture is presented in Figure 1. It consists of: a *processing unit*, *memory*, *peripherals* and *buses*.

Figure 1: Basic scheme of the Princeton hardware achitecture.

A processing unit is dedicated to executing instructions – operations that given an input and produce an output (that could also be an input to another instruction later on). A sequence of instructions that are grouped logically, for the sake of performing some more or less distinct piece of work, form a *program*, which is a single element of the whole software of the computing system. Potentially, the same program could have alternative representations, e.g., being in a form understandable by machine (*machine code*) or being represented in a form that is more suitable for humans (*high-level abstraction code*), expressed with the help of a *programming language*.

The memory is usually represented as an array of cells each of which is able to store one of *two* possible states. These *binary* alternatives could be represented by

"0" and "1", or by any other way to be distinguished, and form a *binary digit (bit)*, of information. All the bits in memory are logically grouped into *words* of some fixed length, that is dependent on the implementation of the hardware architecture. Each word has a unique address, for the purpose of being accessed by the processing unit.

While the memory/processing unit combination is pivotal for a computing system to function, the peripherals provide auxiliary means that make a computer system useful for interfacing with the outside world. Therefore, thanks to the peripherals, a computer system is not a "thing-in-itself", but a tool for solving real-life problems. The peripherals could be classified as *storage*, *input output* components.

Whilst the memory stores the data during an operation phase, the storage components should be able to hold the data that were successfully processed by the computing system even after it will be turned off. Through the input/output components, a computing system receives/sends signals or data from/to the outside world.

All the hardware components mentioned above are connected by the *bus* wiring – communication pathways that provide signal and data transfers.

The processing unit in the Princeton architecture introduced a generic design that is highly influential in computer engineering. Being a general-purpose circuitry, a processing unit needs to be able to interpret properly the data on which the particular instruction operates. Such information should be specified by the instruction itself, therefore, the Princeton architecture implies distinct subsets of instructions for every type of data e.g., one instruction type for discrete mathematical objects (integers) and another one for continuous mathematical objects (floats). This example of having both integer and floating-point arithmetic implemented in the hardware, makes the processing unit circuitry more complex. As a result, the memory and the peripherals were not incorporated into the same circuit with the processing unit.

One of the key aspects of the Princeton design is the memory model. Any memory location is uniquely identifiable and accessible via its address. Additionally, each

memory location may hold instructions, data of arbitrary types or even addresses of other memory locations. It is up to the software running on the processing unit to keep track of/interpret appropriately the contents of each memory location accessed by it. On the other hand, in program representations the addresses potentially can be manipulated using the instructions designed for data processing. Both features described above, not only provide the flexibility in creating complex dynamically changing data structures, but also open a way for a program potentially misbehaving and unauthorized memory accesses. This requires attention in software and hardware design.

However, in Princeton architecture, where there are no separate memories for instructions and data, the bus input/output for instructions and the bus input/output for data intefere with each other. This may be detrimental for performance in scenarios where the processor has to perform a small amount of work on each element of a huge data. This effect (known as the *von Neumann bottleneck* [8]) does not exist in Harvard architcture [180] where the memories and buses for instructions and data are separate.

The Harvard memory model is represented as a combination of an *Instruction memory* and a *Data memory* (see Figure 2).

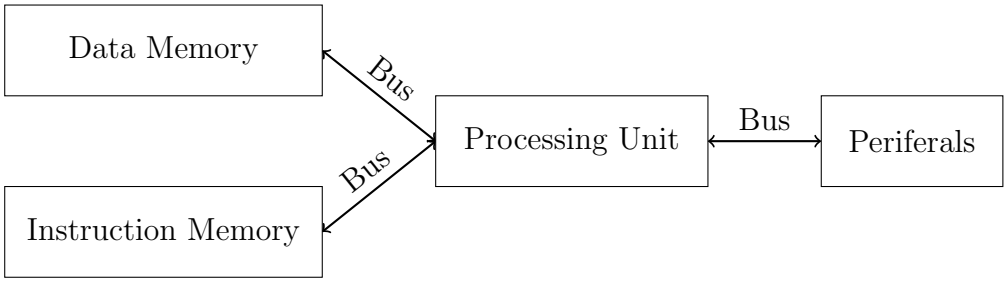


Figure 2: Basic scheme of the Harvard hardware achitecture.

These two memories are independent and do not have to share characteristics, e.g., the implementation technology, the memory address structure, the width of the

word, etc. For example, if it is known that the kind of target application requires lots of processing over small data arrays, a system designer would introduce a larger instruction memory and a smaller data memory – therefore, it might be reasonable to make instruction addresses wider compared to data addresses. A strict distinction between the instruction address space and the data address space requires the data embedded in the code (e.g., the constant values) to be copied to the data memory, which is an obvious performance drawback. On the other hand, the separation between these two memories greatly reduces the potential security hazards for the stored instructions in terms of an inappropriate access.

Such a heterogeneous memory model allows to tweak the hardware for a particular application domain, hence, the Harvard architecture gained a strong popularity for the embedded systems implementations. Even though having a less generic memory model, the Harvard design has some strengths that are important also from a general-purpose viewpoint. Given that the instruction traffic and the data traffic do not have to share the same pathway, an instruction read and a data access can be performed in parallel. Thus, due to the absence of the von Neumann bottleneck, a computing system based on the Harvard architecture can potentially be faster compared to a Princeton-based system for a given circuit complexity.

Let us consider the circuit design principles in more detail.

2.3 Circuit organization

An electronic circuit can be categorized as being *analog*, *digital* or *analog-digital*.

An analog circuit is an electronic circuit that deals with continuously changing *analog signals*. This type of signals corresponds to *continuous* aspects of classical physics phenomena observed in the nature, e.g., electromagnetic field that is considered to extend continuously throughout space. A continuous variability of signal values is proportional to the change in electrical current or voltage that represents

the corresponding signal [2].

Digital signals originate from *discretization* of analog signals – a simplification made by splitting the range of the analog signal in bounded intervals and abstracting away from every part of the signal by representing it just by a single value from the corresponding part. Such a discretization of an analog signal range allows, to some extent, tolerate noise, interference with other signals, etc., and was utilized in *digital circuits*. Since, in many cases, it turned to be more reliable to work with digital signals, digital circuits gained a tremendous popularity. Particularly, an approach of dealing with just two valid voltage areas – the lowest possible (marked as "0") and the highest possible (marked as "1"), is widely accepted by the electronic industry which rely on *Boolean logic* [86].

Typically, most of the electronic components inside a digital circuit, are spent to form the *logic gates*. In a logic gate, the components are arranged in a way to implement some specific boolean function that for a number of binary inputs produces a single binary output. In the voltage range the area between the two extreme areas "0" and "1" is called "forbidden", thus, the corresponding signals are considered to be invalid. The forbidden zone is used to avoid confusing "0" with "1" in a realistic operating conditions where every signal experiences a noise. To tolerate its harmful influence, the voltage bounds for the output signals are more strict when compared to input signals. This is done to anticipate the room for noise by accepting marginally valid input signals, but provide output signals with solid validity.

The analog-digital circuits combine analog and digital approaches. They are very popular for signal amplification, signal conversion from analog form to digital form and vice versa.

The parts of digital circuits can be synchronized or they can work asynchronously.

A *synchronous circuit* has a notion of time by including a part that generates a *clock* signal for coordinating all the actions performed by the circuit. This implemen-

tation of the notion of time is based on the propagation delay of the circuit – the time interval between the moment when the input of the logic gate gets stable, and the moment when the output of that gate becomes stable.

Asynchronous circuits do not have central clock. To coordinate the correct sequence of actions they utilize special signals which indicate that the corresponding action was completed. Such an approach of circuit design not only liberates the performance from the bound imposed by the worst-case scenario, but may also bring power efficiency and allow adaptability to operation conditions (e.g., adaptation of the performance subject to temperature change). Although asynchronous circuits are an active topic of research and development, commercial-off-the-shelf circuits are mostly synchronous so far.

From now on in this thesis, by mentioning a circuit, we assume it to be digital and synchronous if the opposite is not explicitly stated.

2.4 Microarchitectures

In the early days of computing systems, electronic circuits were built of independent electronic components. Therefore, such *discrete circuits* were characterized by huge size, wasteful energy consumption and high direct materials cost. Later, theoretical and practical advancements in semiconductor electronics made it possible to integrate numerous electronic components into a single circuit placed on a small plate ("chip") of semiconductor material. Such *integrated circuits*, called *chips*, have replaced discrete circuits in many fields of electrical and computer engineering due to the rapid growth of functional characteristics, lower cost and lower power consumption. Integrated circuits gained tremendous performance boost and popularity to both embedded and general-purpose computing systems bringing front-edge technology advancements to tiny *microcontrollers* and *microprocessors*.

Although having revolutionized the world of electronics, semiconductor compo-

nents did not change immediately the high-level principles of general-purpose systems. In microprocessors the main memory remained to be placed on separate chips.

However, in the case of embedded-systems hardware, integrated circuits allowed to take an opposite approach. In microcontrollers the processor, the memory and the peripherals are all placed on a single semiconductor plate. This design principle is very suitable for those embedded systems that had minimal requirements for program length and memory size, since there is no need to implement high-end integrated circuits. Also, this allows to make microcontrollers being attractive by cost and energy consumption.

Although taking a lion's share of processing units market for embedded systems [36], microcontrollers were not able to provide enough performance for the systems (e.g., smartphone hardware) that emerged on the frontier between general-purpose and embedded domains. Such systems require a decent computational power while being able to fit into a relatively small energy budget and pocket-size form-factor. Such requirements motivated a technical direction of implementing a higher system integration in so-called *system in package*, *package on package* and *system on chip*.

System in package includes a number of chips assembled into a single chip carrier ("package"). In package on package, chips are stacked vertically during board assembly. System on chip (SoC) integrates in a single chip a number of components, that were traditionally implemented in stand-alone integrated circuits. However, some parts of such systems are still placed off-chip, e.g., the main memory.

Another possible example of merging of general-purpose and embedded domains could be a Graphics Processing Unit (GPU) – originally an input/output component designed for rendering graphics, that has recently evolved towards being an architecture for general-purpose computations.

2.5 Graphics Processing Unit

The term "Graphics Processing Unit" (GPU) was coined [157] by NVIDIA, and naturally, in this thesis we target GPUs designed by this chip-maker. However, the proposed timing analysis approaches can be applied to graphics processors from other vendors as well.

Novel parallel programming models developed for the GPUs brought us to the General-Purpose GPU (GPGPU) [76] computing: the use of GPUs as accelerators for computationally intensive (non-graphics) workloads. The GPUs are widely used to unload the traditional Central Processing Units (CPUs), liberate other resources of a given computing system, and provide an alternative to multiprocessor computers for processing computationally heavy parallel tasks.

Modern GPUs are immensely parallel architectures. NVIDIA GPU (Figure 3) contains several "Streaming Multiprocessors" (SMs).

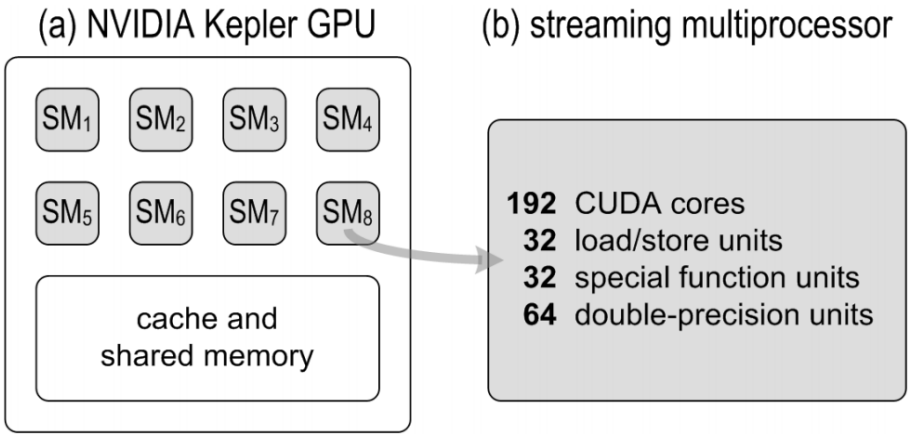


Figure 3: A simplified scheme of the NVIDIA Kepler GPU chip.

Each streaming multiprocessor is a complex manycore in itself, as it includes many

- CUDA cores, for integer and floating-point arithmetic;
- "load/store" units that load data from/store data to the memory subsystem;

- special function units, implementing sine, cosine, square root etc., in hardware;
- double precision (64-bit) units.

The big number of cores is determined by the fact that the GPUs leverage an important aspect of typical graphics workload: data in huge arrays does not have dependency, and therefore, can be processed in parallel. Such *data-parallel* workloads are processed by the GPUs achieving high performance not due to the low processing latency of every single core, but due to the high throughput provided by the whole chip. In this sense the microarchitectures discussed earlier in this chapter can be considered as *latency-oriented* processors while the GPUs are *throughput-oriented*.

Nevertheless, despite the substantial differences in their architectures, computing systems based on any of the microarchitectures discussed in this chapter have a similar operational cycle.

2.6 Operational cycle

Considering program processing at a high abstraction level, a computing system operates in the following way: the code of the program, that contains a series of logical, arithmetic, control, input/output instructions and associated data, is loaded into the memory and the processor performs each instruction in turn. Although most of the software these days is written in high-level programming languages, eventually, all these high-level codes are translated to machine codes – low-level representations of the instructions encoded in binary form processable by computing system circuitry.

Upon receiving the machine code of the instruction, the processor has to recognize from it what kind of actions that instruction requires from its pre-determined functionality to be able to carry out these actions. All the instructions that are supported by the processor form an *instruction set*, which is often processor/architecture-specific. Each instruction in the instruction set has a unique code, that serves as an

identifier of the instruction and is an obligatory component of any machine code. Together with the *instruction identifier field* (also known as the "opcode"), the machine code holds an *instruction operands field* that specifies where the data read/written by the instruction are stored and how this data could be accessed (an *addressing mode*).

The instructions are processed in an *instruction cycle* – an operational cycle that is continuously repeated from boot-up until shut down of the computing system. Simply speaking, the instruction cycle consists of three phases (see in Figure 4): *fetch*, *decode* and *execute*.

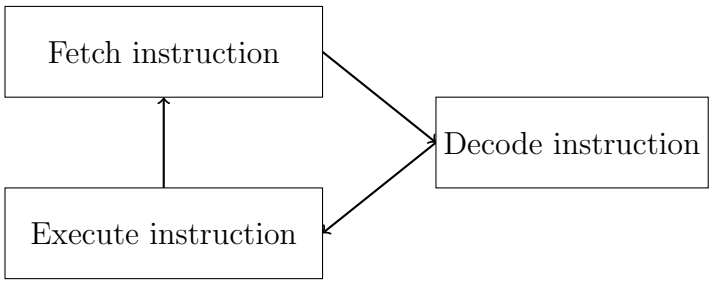


Figure 4: A simplified scheme of an operational cycle.

During the fetch phase, the corresponding machine code is retrieved from the memory and stored in an *instruction register* of the processor – a temporary storage for the instruction to be executed soon. Additionally, other registers of the processor are updated, e.g., the one (also known as *program counter*) that stores the memory address of the instruction to be executed next.

The decode phase stands for the interpretation of the machine code stored inside the instruction register. This is done by examining the instruction identifier field of the machine code for the sake of matching the corresponding instruction from the instruction set of the processor which would allow it to "understand" what kind of actions should be performed to execute that instruction. Then, the instruction operands field should be analyzed in order to get what is the data to process and where it is stored.

During the execute phase, the actual function of the instruction is performed. The decoded instruction is passed as a sequence of control signals to the relevant functional units of the processor. Here, for the sake of simplification, we assume that the execute phase includes accessing the data required for the execution and storing the result of the instruction to the memory – also known as "memory access" and "write back" respectively.

Further, we rely on the essencial terminology and the conventions introduced in this chapter to present the review of the literature in the context of the topic of this thesis.

3 Literature Review

The current state-of-the-art offers a few methods for GPU timing analysis, however, the research literature offers several results for solving related problems. This chapter serves as a brief review of those works.

3.1 Introduction to real-time systems

Real-time computing is usually defined as a study of hardware and software aspects of systems that have time constraints (e.g., a computer that controls an autonomous driving vehicle). In this work we pay attention to software programs that must execute and give response during a particular time window. On the other hand, a non-real-time system is one that has no deadline, even if fast response or good performance is appreciated.

A number of definitions of real-time systems cover a broad spectrum of computing systems. A definition of Randell et al. [166] is the following: "A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment."

Young defines [185] a real-time system as "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period."

The Oxford Dictionary of Computing states [164] that "Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness."

Burns et al. emphasize [34] a pivotal aspect that distinguishes real-time systems from other systems: "the correctness of a real-time system depends not only on

the logical result of the computation, but also on the time at which the results are produced."

Real-time computation is said to be failed if it is not completed before its *deadline*, regardless of the amount of work that a computer system performed during the corresponding period of time. If the system in consideration tolerates no missed deadlines at all (e.g., possibly because of catastrophic consequences), then it is called *hard real-time*. Otherwise, the term *soft real-time* for the system is used. Where to put the borderline between hard real-time systems and soft real-time systems greatly depends on the applications domain, but for system development, being hard real-time means satisfying much stricter timeliness guarantees.

The purpose of real-time computing is to execute *tasks* in a timely manner. A task is an abstract entity of execution that can be substituted by those of "real-world" computer systems (e.g., a process, a thread, etc.). Each task has resource requirements. All tasks require some execution time on a processor and also a task may require a certain amount of memory, access to a bus, etc. Sometimes, a resource is only used by one task, but in other cases, resources are shared, which may require some control over the access to the resource. The same resource may be exclusively or non-exclusively accessed, depending on the operation to be performed on it, e.g., memory object (writing is exclusive but reading is non-exclusive).

The *release* time of a task is the time at which all the data, that are required to begin executing the task, are available and the *deadline* is the time by which the task must complete its execution. If a time-critical task does not successfully complete by its deadline, a *timing fault* occurs. In such situation the result of the task execution becomes of little or even no use.

In real-life systems, the goal of meeting all the deadlines is challenging because of dynamic factors (e.g., variations in processing times) that occur because of the system indeterminism imposed by sophisticated hardware and software components.

One way to deal with these difficulties is presented in an approach that trades off result quality to meet execution requirements via *imprecise computation* [42]. The basic idea underneath the imprecise computation is to process first a mandatory workload and only then catch up with less important work. This principle of prioritizing important part of work at a price of leaving non-mandatory part to be potentially unfinished is implemented via augmenting traditional task model that was presented above. The system designer has to structure a time-critical task to contain a *mandatory subtask* and an *optional subtask*. To get an acceptable result of a task, its mandatory subtask has to be processed before the task's deadline. The further execution of the optional subtask is supposed to refine the intermediate result obtained by the mandatory subtask. If the optional subtask will complete successfully, the refined result is called *precise* and is assumed to have a zero-error. Otherwise, the *imprecise* intermediate result is promoted to be the final result of the task and is usually associated with some degree of error.

Thus, imprecise computation prevents timing faults by providing an approximate result of a reasonable quality whenever obtaining an exact result in time is not possible. This approach is suitable for applications featuring *monotonicity* – a property which requires that the quality of the intermediate result does not decrease with increasing processing time. This property is common for many algorithms in the areas of sorting, heuristic search, numerical computations, database query processing, etc.

In real-time systems domain, task is described by a piece of code that is executed in a repetitive manner. Every distinct execution of that code, say of some task τ_i, corresponds to one more task instance called *job* $J_{i,j}$ (a job j of a task τ_i) being released. In terms of repeatability, tasks may be categorized in three different families: *periodic*, *sporadic* or *aperiodic*.

According to the periodic task model [126] a task τ_i is periodic if it is released periodically, let us say every T_i time units (the respective *period* of the task τ_i). The

periodicity constraint requires the task to run exactly once every period, but it does not require that the task be run exactly one period apart. Quite commonly, the period of a task is also its deadline. Task invocations usually are also called *job releases* or *job arrivals*. The worst-case execution requirements C_i is the maximum amount of time needed for execution of each job that was generated by τ_i.

The task is sporadic if it is not periodic, but may be invoked at irregular intervals [146]. In this context, T_i denotes the respective *minimum inter-arrival time*. Sporadic tasks are characterized by an upper bound on the rate at which they may be invoked.

Aperiodic tasks are defined to be not periodic and have no upper bound on their invocation rate.

To measure how the collection of n tasks assigned to a single processor utilize this processor, the system *utilization* U is defined as:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

This definition of the uniprocessor utilization is made subject to an assumption that the processor is allowed to execute at most a single task at a time, and a task (as well as its jobs) cannot execute on two or more processors simultaneously. For the case of multiprocessor system that includes m identical processors, the definition of the computing system utilization can be extended as follows:

$$U = \frac{1}{m} \cdot \sum_{i=1}^{n} \frac{C_i}{T_i}$$

The scientific discipline of real-time systems considers two problems (i) *schedulability analysis* and (ii) *Worst-Case Execution Time* (WCET) analysis.

An objective of WCET analysis is to derive the values of the worst-case execution times C_i for every task τ_i of a task set under consideration. Then, these values of C_i

are submitted to the schedulability analysis as an input. The goal of the schedulability analysis is to find out whether the task set is *schedulable*.

A collection of tasks is schedulable by a scheduling algorithm SA if this algorithm ensures that the timing constraints of all tasks are met.

A task schedule is said to be *feasible* if all the tasks start after their release times and complete before their deadlines.

The utilization bound UB_{SA} of an algorithm SA is the maximum number such that if $U \leq UB_{SA}$, then all tasks meet their deadlines when scheduled by SA.

A schedule may be prepared before (*offline scheduling*), or obtained dynamically (*online scheduling*). Offline scheduling involves scheduling in advance of the operation, with specification of when the periodic tasks will be run and slots for sporadic or aperiodic tasks in the event that they are invoked. In online scheduling the tasks are scheduled as they arrive in the system. The corresponding algorithm should be as fast as is necessary to leave sufficient time for tasks to meet their deadlines.

The schedule of tasks may be *preemptive* or *non-preemptive*. A schedule is preemptive if tasks can be interrupted by other tasks and then resumed. This allows higher-priority tasks to preempt lower-priority tasks (whether these priorities are static or dynamic), in order to meet deadlines. Preemption allows the flexibility of not committing the processor to run a task through to completion once we start executing it. By contrast, once a task is begun in a non-preemptive schedule, it must be run to completion or until it gets blocked over a resource.

Examples of scheduling algorithms with a rich literature of associated schedulability analyses include Rate-Monotonic (RM), Earliest-Deadline-First (EDF) [126] for uniprocessor systems. On the other hand, the majority of scheduling problems on systems with more than two processors are NP-complete [43], thus for their solving some heuristics are usually utilized. A lot of them are based on uniprocessor scheduling. In such cases the problem of developing a multiprocessor schedule consists of

two subproblems. The first one is about assigning tasks to a processor. The second subproblem is about running uniprocessor scheduling algorithm for each processor and the corresponding task subset, in order to meet the respective deadlines. Often, in engineering practice multiple iterations of these two steps (in a loop) are performed until a feasible schedule is found.

The scheduling approach described above is termed *partitioned*. Its main strengths are the simplicity (stemming from the decomposition to multiple uniprocessor scheduling problems) and the ability to use the state-of-the-art uniprocessor scheduling algorithms known for their efficiency. The main weakness of partitioning is that the utilization bound of such approaches is inherently limited to 50% or less [159].

At the other end of the classification spectrum from partitioning, lies global scheduling [120], [51]. Algorithms of this category employ a single run-queue for all ready tasks. At any time instant, the highest-priority ready tasks execute, each on a different processor. This implies that task *migration* is allowed: each task may execute on any processor and in fact, it may migrate to another processor halfway through its execution.

Policies familiar from uniprocessor scheduling have been extended to global scheduling as well (global EDF, global RM) but their respective utilization bounds are much lower than even 50 % of their partitioned versions. Some other global scheduling algorithms (such as those from the proportionate fair (Pfair) [13] family) have a utilization bound of 100% [3], but are impractical from an implementation perspective because of the high number of preemptions.

Consequently, researchers have turned to semi-partitioned schemes, which try to combine the best of partitioned and migrative scheduling. Under such schemes (e.g., EDF-WM [100], EDDP [106], NPS-F [29]), only a few tasks migrate, in a very controlled manner. This allow efficient processor utilization (and utilization bounds above 50%) without the overheads of global scheduling.

The scheduling algorithms mentioned above were proposed in the context of the traditional task model described in this section. However, a principal factor that influences the success of one or another scheduling theory to a particular real-life application, is whether the underlying task model fits to the corresponding application domain. Naturally, the traditional task model, briefly described above, is not a "silver bullet". Hence, the real-time research community proposed other models, which we are going to discuss next.

3.2 Rate-Based Execution model

Jeffay et al. presented [97] a generalization of the sporadic task model [146] and the periodic task model [126]. Unlike these two models that characterize a task under consideration using an exact value or a lower bound on the inter-arrival time of its jobs respectively, the authors considered an expected arrival rate of the jobs. In other words, the researchers do not make assumptions about the time instants at which the jobs arrive. Instead, they assume that the jobs arrive at a given average rate, while the corresponding distribution of the arrival time instances is arbitrary. Thus, they called this approach Rate-Based Execution (RBE) model. The motivation of RBE is supported by an observation that in many applications with timing constraints (e.g., digital signal processing or multimedia systems) the arrival of the events does not match well enough neither periodic nor sporadic task models. For instance, the video streaming applications are usually characterized by arbitrary instantaneous reception rates of video frames, while the respective average rates are kept pre-defined.

Therefore, the RBE task is defined through the following parameters:

- the length of the time interval that was chosen for the rate characterization;
- the maximum number of task instances (jobs) arrived per time interval specified above;

- the relative deadline of the task instance;
- the worst-case execution requirement of the task instance.

The authors observed that in the context of EDF-based scheduling, the feasibility of RBE task sets is a function of the distribution in time of the respective deadlines. Taking into account, that applications usually have some level of control over the deadlines (e.g., the deadline assignment is done by the operating system), the researchers argued that the real-time system designer is supposed to have more control over the operating systems rather than over the external processes that provide the system with the workload. Thus, the deadline-based scheduling is more appropriate to the RBE task sets when compared to priority-based scheduling where the feasibility of RBE task sets is a function of the rate at which the respective jobs arrive.

Focusing on the event-driven real-time systems, the applicability of the RBE model to the signal processing workloads was demonstrated [71] by Goddard et al. Earlier, Jeffay et al. have motivated [96] the use of the RBE model for the multimedia computing.

The above approaches consider non-parallel (i.e., sequential as in Section 3.1) tasks. Although the sequential task models simplify the complexitiy of the timing analysis and the scheduling, these models are restrictive for the most of the modern commercial-off-the-shelf hardware since they do not allow to exploit underlying parallelism properly. Thus, to take an advantage of the potential parallelism, the community was developing more adequate models of the tasks.

3.3 Parallel task models

Parallel hardware architectures allow to decrease the execution time of the tasks and improve the utilization of the processors by splitting the tasks into smaller entities of computation (e.g., threads) that can be executed in parallel on different computa-

tional units (e.g., cores). Although, this led to shorter response times and improved schedulability, the problems of timing analysis and scheduling are getting one more dimension in terms of complexity. To handle this execution paradigm, the literature offers techniques and models for *parallel tasks*, implemented as multiple parallel threads. In this context, there are two common scenarios:

- the threads are organized in a "gang", where all the threads execute or become idle all together in parallel on different computational units (the *gang model*);
- the threads tend to perform execution independently and synchronize at the beginning and at the finishing of the execution (the *independent thread model*).

3.3.1 Gang model

Ousterhout et al. introduced [161] the gang model for executing multiple threads that frequently interact with the help of a message passing interface (implicit synchronization) or synchronization barriers. Rather than schedule individual threads, this model considers a gang to be the schedulable entity. The idea behind the gang scheduling is to make the threads within a gang start and stop simultaneously for the sake of reducing processor idling and context switching overheads.

Usually, in real-time systems the tasks are recurrent. Each single launch of the corresponding code leads to the release of one more *job* of respective task. In other words, job is a logical abstraction that corresponds to a single launching.

For the parallel tasks, Goossens et al. presented [74] a categorization of parallel jobs, according to the variance over time of the degree of intra-task parallelism, that includes three types: *rigid jobs*, *moldable jobs* and *malleable jobs*.

Definition 1. *(Rigid job)*

A job $J_{i,j}$ is said to be rigid if the number of parallel threads of $J_{i,j}$ that must be executed synchronously is task-static and defined externally to the scheduler, a priori

and does not change throughout the execution.

Definition 2. *(Moldable job)*

A job $J_{i,j}$ is said to be moldable if the number of parallel threads of $J_{i,j}$ that must be executed synchronously is defined by the scheduler and does not change throughout the execution of the job (job-static). Therefore, the scheduler may take decision on the number of created threads regarding, for instance, the current workload on the platform.

Definition 3. *(Malleable job)*

A job $J_{i,j}$ is said to be malleable if the number of parallel threads of $J_{i,j}$ that must be executed synchronously can be modified by the scheduler during the execution of $J_{i,j}$.

In the literature review of this thesis we rely on this terminology to describe the related work.

Kato et al. applied [99] the Earliest-Deadline-First (EDF) [126] scheduling policy to the gang scheduling scheme. The authors presented schedulability analysis of Gang EDF by identifying the interference bound for the deadline miss and by deriving the schedulability test based on the one for the Global EDF [14]. For this integration of the gang scheduling and the Global EDF, the authors assumed that the number of threads, and therefore, the number of processors needed for the execution of any job $J_{i,j}$, is set by the system designer beforehand. This assumption complies with the definition of *rigid job*[2] [74], which poses difficulties in applying state-of-the-art single-threaded scheduling schemes. The problem is the following: for its execution, the rigid job $J_{i,j}$ needs exactly n_j processors available, where n_j is set a priori. Hence, this principle of specifying n_j statically, can lead to some form of the priority inversion that will happen when the higher-priority rigid job does not have enough processors available to run, while the lower-priority rigid job does.

[2]Note, that Kato et al. used some different terminology and called [99] their jobs “moldable”.

Goossens et al. extended [74] the definition of rigid job to *rigid task*, that is such that it holds rigid jobs only, but all these jobs do not necessarily require the same number of processors to execute. The authors extended four fixed-priority scheduling schemes to be applicable for rigid jobs and rigid tasks, namely: Parallelism Monotonic, Idling, Limited Gang, and Limited Slack Reclaiming. Considering a fixed task priority assignment which specifies the priority of every task (and all its jobs) beforehand, they provided exact schedulability tests for these scheduling policies.

Although rigid task model causes more deterministic behaviour, it hurts the schedulability. Hence, the real-time community demonstrated an interest in an idea of giving to the scheduler the freedom to decide how many threads will be used for the execution of the parallel job under consideration. Although for such *moldable* jobs [74] the scheduler can adjust the number of threads to the number of processors available, this *degree of parallelism* [136](the number of threads) has to be kept unchanged throughout the execution of the job. The interest in moldable jobs is present in the community for a long time. Han et al. provided [82] an off-line method for deriving the number of threads for each job from a finite set. The authors considered the preemptive fixed-priority scheduling and proved that for such a task model it is NP-hard. Hence, they proposed a heuristic-based algorithm for the task-partitioning on two processors.

Liu et al. also considered [136] static scheduling, but they did not put any constraints on the number of processors in the system. The authors also addressed the fact that the parallel execution causes some processor time being wasted on inter-processor communication and synchronization. Their model considers independent jobs, where each one requires some amount of processing time that can be spent by available processors via executing that job in parallel subject to the upper-bound on the respective degree of parallelism. Hence, every job is characterized by a number of parameters including the ready time, the deadline, the maximum degree of paral-

lelism and the multiprocessor overhead factor. The authors divided the time between the earliest job ready time and the latest job deadline into the time intervals with the help of intermediate ready times and intermediate deadlines used as simple preemption points. Then, the authors considered the processor time allocation problem subject to an assumption that the parallel processing overhead has to be a linear function of the degree of parallelism. This assumption allowed them to formulate the processor time allocation problem as a linear programming optimization problem.

Even though preemptive scheduling is more flexible, Manimaran et al. argued [143] that the schedulers of such kind usually suffer from a serious overhead that occurs because of the context switching triggered by every preemption. Thus, they considered a non-preemptive dynamic scheduling in a way to keep the overheads under check. Their EDF-based approach consists of an off-line stage – where the tasks with known periodicities are parallelized and analyzed in terms of schedulability; and an on-line stage – where those tasks are scheduled together with the aperiodic tasks. The authors pointed to the potential timing anomaly for the case when some job executes faster than in its worst-case scenario and provided a circular queue-based mechanism to partially mitigate this issue.

The gang scheduling would demonstrate improved schedulability if the scheduler could adjust the job‘s degree of parallelism at run time during the execution of that job. Such malleable jobs being released by sporadic tasks were investigated by Collette et al. [45]. The authors considered sporadic implicit-deadline tasks on an identical multiprocessor platform. In their model the tasks are scheduled globally subject to an assumption of *work-limited job-parallelism* which is another form of parallelism restriction also discussed by Liu et al. [136] and Manimaran et al. [143]. The intuition behind this assumption is that even though increasing the number of processors from p to p' will provide a faster execution of a parallel job, this job will not run $\frac{p'}{p}$ times faster than it runs on p processors. Additionally, the returns are diminishing with every

additional processor. Subject to these assumptions, the researchers presented [46] a proof which states that analyzing the feasibility of the task set has a linear time-complexity with regards to the number of tasks. Based on that proof, the authors proposed an optimal scheduling algorithm, an exact feasibility utilization bound and a technique for limiting the number of migrations and preemptions.

Although malleable jobs provide the most flexible way of gang scheduling, they pose a serious challenge in terms of implementation. Modifying the number of the threads allocated to a job at run-time is not that straightforward and would also require a substantial overhead. Berten et al. proposed [27] a sufficient schedulability test for a special kind of parallel tasks. In their model, each task is supposed to be represented as a sequence of segments with a precedence constraint, thus the segment $s + 1$ cannot start its execution until the segment s will finish. The scheduler is supposed to make the decision on how many threads the segment should be executed subject to the maximum degree of parallelism of that segment. The idea of the approach is to use the time-instants between the consecutive segments for deciding the number of threads for the execution of the following segment, but also to forbid the scheduler to change this number during the execution of the respective segment.

The characterization of rigid, moldable and malleable jobs presented above is applicable to the case of gang scheduling because it is required to schedule threads synchronously. For the case when there is a need only for a coarse-grain interactions between threads, the following models were invented.

3.3.2 Independent thread models

Lupu et al. presented [140] a constrained deadline model of periodic parallel tasks processed on an identical multiprocessing platform. The parallelism in this model is expressed in the following way: a task includes a set of "subprograms" that can be executed in parallel. Consequently, the task has the potential to progress in execution

upon several processors concurrently. While the period and a relative deadline are the features of the task, the worst-case execution time is determined via a set of execution requirements – each one for a respective subprogram. Studying the schedulability of such a model, the authors distinguish between offline entities of computation (tasks, subprograms) and their runtime instances (*processes, threads* respectively) that should be actually managed by the priority-driven preemptive scheduler. The possible schedulers are assumed to fall in one of the categories: *hierarchical* and *global thread schedulers*. The hierarchical schedulers should firstly deal with the process and only after that schedule the threads within each process. The global thread schedulers are supposed to grant individual priorities to every single thread regardless of its process or the respective offline instances. Furthermore, each one of these two categories includes multiple schedulers that are representing different scenarios regarding whether the priority of a runtime instance is based (or is not based) on the corresponding offline entity. The authors developed an exact schedulability test for each category.

In the case of the model discussed above, the runtime instances are implicitly synchronized by release time. Other than that, the only explicit synchronization could be the common deadline. However, the fork-join model [47] provides more control, in this sense. This model allows the designer to set up a parallel application by defining the points in the corresponding code where execution may branch off in parallel (*fork*). After that, these multiple parallel executions has to be merged at a subsequent point that was set up by the designer to resume sequential execution (*join*). By combining this model with a traditional model of real-time recurrent tasks [126], the real-time job can be considered as a sequence of code segments, each one representing either sequential or parallel stage of execution. A sequential segment of code is executed by a single thread (*master thread*). Then, this thread spawns multiple parallel threads to run the parallel code segment. All these threads synchronize at the end of the

segment and the master thread resumes its execution.

Lakshmanan et al. considered [112] a fork-join model subject to the following restrictive assumptions:

- the execution of each task has to be characterized by a strict alternation between the sequential segments of the code and the parallel segments of the code;
- all the parallel segments of the code are designed for the same constant number of threads to be spawned;
- this number should not exceed the number of processors in an identical multi-processing platform;
- an execution requirement is a feature of the code segment, thus, all the threads that belong to the same parallel segment have equal worst-case execution times.

For each task, the researchers introduced a "master string" that initially holds the elements (threads) of all code segments with respect to their precedence. If the segment under consideration is sequential, the master string will include the respective master thread. In the case that the segment is parallel – the master string will initially hold just one of the spawned threads (e.g., a thread that has the "lowest" identifier). The construction of the master string has to be done with respect to the deadline of the task. Therefore, this deadline should be greater than or equal to the total execution requirement of the master string. The approach presented by the authors is based on an idea that the interference of every single task with other tasks from the task system should be as low as possible. To achieve this goal, the researchers consider whether the execution requirement of the master string is strictly less than the deadline of the task, and if it is the case, they "stretch" the master string. The process of stretching is done by "inflating" the master string with the threads performing parallel sections. This is done, until the execution requirement of the master string would be equal to

the deadline of the task. The equality is achieved by permitting a split of the thread between the processors subject to the restriction that the parts of the split thread cannot be processed in parallel.

Therefore, thanks to the stretching, the master string occupies a single processor, and correspondingly, the threads of the master string do not interfere with the other threads of the parallel code segments. Those remaining threads are assigned to available processors and have to be executed before the "artificial deadline". This deadline is determined by the precedence constraint of the code segments in the master string. It is caused by the artificially stretched parallel segment in a master string that imposes execution time restriction on the other threads of that segment. The authors proposed to partition these threads among available processors according to the algorithm presented [62] by Fisher et al. Then, locally on every processor, the threads are scheduled according to the Deadline Monotonic algorithm [120].

The approach presented above decreases the interference at a cost of limiting the parallelism by achieving 100% utilization of the processor that executes the master string. However, in general case, it forces thread migration, which poses difficulties for the practical implementation of the approach, especially on the general-purpose platforms where the migration is allowed only on the level of "heavyweight" threads or OS processes. Fauberteau et al. proposed [58] to eliminate thread migrations for the same restricted fork-join model of parallel real-time tasks. Thus, they fill the master string only with the complete (integer number) parallel threads.

The approach presented [169] by Saifullah et al. is based on the model described above. The authors relaxed the assumption about the equal number of threads for all parallel segments. Thus, every parallel segment is allowed to spawn an arbitrary number of threads regardless of the number of available processors. Still, all threads, of a given segment, have to be characterized by equal worst-case execution requirement, as each other. The researchers also dropped the assumption about the obligatory al-

ternation between sequential and parallel segments. Thus in their model, subsequent parallel segments are allowed, subject to the requirement that all the threads of each segment are synchronized at the end of the segment.

Similarly to the work [112] discussed above, the researchers utilized the concept of artificial deadlines introduced [173] by Sun et al.[3] Saifullah et al. proposed to decompose every implicit-deadline parallel task into a chain of constrained-deadline sequential subtasks. In particular, an artificial deadline is derived for every code segment, and is assigned to every thread that executes that segment in the following way.

Every segment of the task has to be classified to be either a "light segment" or a "heavy segment". This is done by comparing the respective number of threads against the following ratio:

- the worst-case execution requirement of the task with regards to
- the difference between the period and the sum of the execution requirements over all segments of the task.

If there is no heavy segment in the task under consideration, the slack should be distributed among all the segments proportionally to the respective worst-case execution requirements.

For a task that has some heavy segments, the distribution of the slack should be done as follows. The light segments are assigned no slack, thus, their relative deadlines are equal to the respective worst-case execution times. Therefore, the whole available slack is distributed among the heavy segments such that, all these segments should have an equal *density* (the ratio between the worst-case execution requirement and the deadline).

[3]In this work, the authors presented synchronization protocols for distributed real-time systems composed of periodic tasks. The tasks under consideration were supposed to be the chains of subtasks with precedence constraints. The artificial deadlines were used to manage the release of subtasks being executed on different processors and ensure that the precedence would be satisfied.

The decomposition strategy discussed above, allowed the researchers to apply an established schedulability analysis of independent sequential sporadic tasks on a multiprocessor to the model of periodic parallel real-time tasks presented above. This was done for the sake of proving the resource augmentation bounds for the global EDF and partitioned Deadline Monotonic scheduling algorithms subject to an assumption about zero-cost preemptions.

Nelissen et al. modified the Multi-Threaded Segment model [169] developed by Saifullah et al. In the modified model, every parallel task is represented as a sequence of segments. Every segment consists of parallel threads, each one characterized by a thread-specific worst-case execution requirement. The researchers proposed a greedy algorithm that at each iteration assigns an offset and a deadline of a segment under consideration. The algorithm is aimed at minimizing the number of processors needed, by maximizing the number of segments with at leas the average density of the task. Once changed, from the initially assigned default value which is calculated based on a rough heuristic, the intermediate deadline is not allowed to be updated any more. This provides computational tractability to the approach, but also makes the order in which the segments are analyzed crucial. The authors motivated their work, by showing that their approach dominates the decomposition algorithm [169] of Saifullah et al., by providing a smaller requirement on the number of processors. The proofs of the effectiveness of the proposed solution, with respect to the number of processors needed, are also presented.

Based on the synchronization intensity of the workload under consideration, Feitelson et al. discussed [59] pros and cons of the independent thread model and the gang model. If the time between the synchronization points is relatively long when compared to the overall execution time, the independent thread model is favorable. Otherwise, the gang model is more suitable.

Even though the independent thread model is usually considered to be a good

fit for the mainstream computing, it requires a propper support for the inter-task dependencies. Thus, the community was considering more generic approaches.

3.4 Graph-based models

The real-life workloads often cannot be adequately described by the models based on the independent threads only. The reason is that the scenario where the input of a job of one task (τ) is dependent on the output of a job of another task (τ') is very common. The inter-task dependencies can be considered from the producer/consumer perspective, such that a job of task τ' *produces* the data which are then *consumed* by the job of task τ. These producer/consumer relationships can be described in the form of graphs, where every single data dependency is expressed as a directed edge going from the vertex representing a producer entity to the vertex representing a consumer entity.

3.4.1 Processing Graph Method

One of the models expressing data dependencies is the Processing Graph Method (PGM) [98]. In PGM, the data production and the data consumption are modeled through the *tokens* that are traveling through the graph in the direction from the producer-vertex to the consumer-vertex. The corresponding edge is characterized by the following parameters:

- the number of tokens produced for the given consumer every time the job of the specific producer completes;
- the overall number of tokens consumed for every single execution of the job of the specific consumer;
- the lower bound on the number of tokens available on a specific edge before the job of the respective consumer may start its execution.

These parameters are used for the implementation of the following high-level principle. When the entity of execution (represented by the vertex) consumes sufficient data, it executes the respective job from start to end, without synchronizing with other nodes. Upon completing it produces the data for the next consumer.

Goddard developed [72] an approach for the transformation of PGM graphs into rate-based tasks by combining the signal processing graphs and the real-time systems domains. The author characterized the real-time properties of the signal processing graphs introducing the notion of *latency* – the time interval between the moment when a signal sample is received from the sensors as an input, until the moment when the graph outputs the processed signal. The researcher distinguished between the latency caused by the topology of the vertexes in the graph and the latency imposed by the scheduling/execution of the respective jobs. The author argued that representing a PGM vertex as a task in the RBE model is more practical when compared to the modeling of the respective vertex as a periodic or sporadic task. This statement is motivated by the analysis showing that for the given signal processing scenarios, the execution of the job according to its rate specification imposes less latency compared to the interpretation of its execution in the context of the classical sporadic model. This required the development of the techniques for mapping the graph vertexes to the RBE tasks, computing execution rates for every task and formulating the conditions for the verification of the EDF-schedulability for the produced RBE task set.

The approach discussed above is focused on uniprocessor computer systems. Based on this work, Liu et al. addressed [132] the globally scheduled multiprocessing scenario extending the previous approach in the following way. The authors assumed an acyclic PGM graph as an input and posed a problem to output a corresponding sporadic task system. This was done through the intermediate step of representing the PGM graph as the RBE task system that is based on the work of Goddard [72]. In these two works, the sophisticated notion of precedence between the graph vertexes

is addressed in different ways. Goddard used [72] a ready queue to store the jobs in the earliest deadline first order. Instead, Liu et al. redefined [132] the releases and the deadlines. Afterwards, the researchers transform RBE tasks to sporadic tasks and derive the global schedulability condition. The authors showed that the acyclic graph precedence can be ensured under Global EDF scheduling on multiprocessors without utilization loss for the ordinary sporadic task system.

Next, we are going to discuss models of parallelism that have the absence of cycles in the corresponding graphs as a central assumption.

3.4.2 Directed Acyclic Graph (DAG) model

A DAG task is structured as a set of execution portions (subtasks) that are represented as the graph vertexes. The precedence constraints between the subtasks determine the task execution flow and are expressed with the help of the directed edges. Although the precedence may be relatively sophisticated (i.e., multiple input or multiple output edges for a single vertex) a cyclic precedence is not allowed. The subtasks are classified according to their inputs and outputs. A *source* subtask has no input edges. Accordingly, the *sink* subtask has no output. The execution of a DAG task starts from the activation of its source subtasks. The task terminates when the execution flow reaches a sink subtasks.

Similarly to a sequential task, a DAG task has a relative deadline and a period. The task is supposed to generate an unbounded number of task instances (task jobs) that are separated by a minimum or an exact inter-arrival time. Each such task instance consists of a set of jobs of the subtasks subject to the inter-subtask precedence constraints of the respective DAG. Every subtask has a specific worst-case execution requirement, while an execution of the whole task instance has to satisfy the relative deadline of the task. Although a DAG task allows parallel execution, the inter-subtask parallelism is not obligatory. The decision on whether the DAG has to be

executed concurrently or sequentially as a chain, is supposed to be taken by the scheduler. Hence, the WCET of the DAG task is the sum of the worst-case execution requirements of all its subtasks.

Typically, the topology of the DAG allows the scheduler to choose among many options on the scheduling order of the subtasks, however, in the real-time community, the most popular approaches are based on the concept of the *critical path*. This critical path for a given DAG task is the path from the source subtask to the sink subtask which requires the longest sequential execution assuming that there is no restriction on the hardware resources of the computer system that is responsible for the processing. Based on this concept, one can consider the DAG task as a chain of the subtasks (*critical subtasks*) that belong to the critical path. These critical subtasks have to be executed sequentially, while the other non-critical subtasks can be executed in parallel with the critical subtasks.

The problem of scheduling DAG tasks on multiprocessors was considered [18] by Baruah et al. The authors assumed that all the subtasks in a sporadic arbitrary-deadline DAG task are released simultaneously to execute according to the precedence constraints until a given relative deadline. The task was assumed to be characterized, among other things, by two timing characteristics:

- the length of the critical path;
- the total worst-case execution requirement.

The researchers pointed that for the case when the inter-arrival time of the task is greater than or equal to the deadline, the problem of scheduling the respective subtasks reduces to the makespan minimization problem [78]. Thus, list scheduling [77] – the polynomial approximation algorithm that bounds deviations from the optimal solution, can be applied. Considering the hardness result [174] presented by Svensson, that stated that improving the previous algorithm is NP-hard, Baruah et al.

concentrated on the case when the deadline is greater than the inter-arrival time of a DAG task. Therefore, more than a single instance of the same task may be active in runtime at a moment. For such case, the authors argued that the efficient state-of-the-art algorithms and schedulability tests are not applicable in general case.

The researchers proved that the scenario when the instances of the DAG task are released strictly periodically ("synchronous arrival sequence") does not guarantee an occurrence of the worst case. Therefore, studying this scenario is not enough for discovering schedulability properties. The authors considered EDF scheduling and presented two tests for checking whether the corresponding algorithm would be able to schedule the sporadic task under consideration on identical multiprocessors subject to all the deadlines. One of these sufficient schedulability tests has a polynomial runtime complexity, while another one is pseudo-polynomial.

Li et al. extended [121] the work discussed above for the case of multiple sporadic DAG tasks under the Global EDF scheduling policy. At each time instant the algorithm is supposed to schedule the vertexes from the task instances with the earliest deadlines subject to the requirement that the predecessors of these vertexes have finished their execution. The authors provided performance guarantees in the form of a resource augmentation bound: a scheduling algorithm SA provides a resource augmentation bound b if SA can schedule a task set on m processors of speed b given that a feasible schedule exists, provided by some hypothetical optimal scheduler, for m unit-speed processors. However, Fisher et al. have showed [63] that for two or more processors a feasible sporadic task system may be found such that it cannot be scheduled correctly by any online deterministic algorithm. The impossibility of the optimal online scheduling implies that for any task model that generalizes the sporadic task model, an optimal multiprocessor scheduling is impossible. Therefore, if it is not possible to claim that a task set under consideration can be potentially scheduled by some "ideal" algorithm, an existence of the resource augmentation bound

does not mean the schedulability. Thus, Li et al. also introduced a *capacity augmentation bound* – a resource augmentation bound that provides a schedulability test. A scheduling algorithm SA with a capacity augmentation bound b can schedule a task set on m processors of speed b subject to the requirement that the total utilization of the task set is less then or equal to m and the critical path of every task is less then the respective deadline. Based on this distinguishing between the resource augmentation bound and the capacity augmentation bound, the authors proved two bounds for Global EDF scheduling of sporadic DAG tasks on an identical multiprocessor:

- a resource augmentation bound for arbitrary-deadline tasks;
- a capacity augmentation bound for implicit-deadline tasks.

Bonifaci et al. considered [32] the same model as the work discussed above, but without the restriction on the deadlines. In other words, the authors do not require that all the jobs of an instance of a DAG task should finish before the next instance of that DAG task can be released. Considering the Global EDF and the Deadline Monotonic scheduling, the authors proved the resource augmentation bounds[4] for these algorithms. They also presented polynomial and pseudo-polynomial time complexity tests for determining whether a DAG task set can be scheduled by the algorithms under consideration.

In terms of timing characteristics, the works already discussed in this subsection did not take into account the internal structure of DAG tasks under consideration. The vertexes of the DAG are characterized by their WCETs and the timing parameters inherited from the DAG task, such as an offset, a deadline and an inter-arrival time. Thus, while the precedence between the vertexes is respected, all of them nevertheless share the same deadline of the corresponding DAG task. Intuitively, it would be easier to avoid the DAG task deadline miss, by introducing the concept of subtask deadline – the intermediate deadline that corresponds to the respective vertex of

[4]In this work, the researchers used a term “speedup bound”.

the DAG. Even though a subtask deadline should be earlier than the task deadline, missing of this intermediate deadline means a deadline miss of the whole DAG task.

On the other hand, the scheduling of DAG tasks poses more challenges compared to the scheduling of sequential tasks. A release of a DAG task instance activates only a job of the source vertex, while the jobs of other vertexes are kept in a ready state waiting for the activation. A ready job can be activated only after all its predecessors' jobs complete. Therefore, the scheduling of the predecessor jobs determine the activation time of a ready job. This dynamic feature of the DAG scheduling imposes that at a given moment in time the scheduler is not aware about the activation times of the jobs of the successors' subtasks.

One of the approaches to deal with such complexities is to convert the DAG task set into a task set that conforms to one of the sequential real-time task models. In particular, this means a transformation of every DAG task into a set of independent sequential subtasks. To satisfy the precedence constraints imposed by the original DAG topology, the sequential subtasks are assigned intermediate offsets and deadlines. Based on these timing attributes, the independence between the jobs of the subtasks is ensured and the scheduling decisions can take these attributes into account. Unfortunately, the simplification gained by using this approach comes at a price of losing the generality provided by the DAG. Adding timing characteristics to the sequential subtasks restricts the schedulability and increases the level of pessimism of the model.

Qamhieh et al. extended [165] the stretching algorithm, presented by Lakshmanan et al. in the context of fork-join tasks, for the case of periodic DAG tasks running on a homogeneous multiprocessor system. Similarly to the original stretching algorithm [112], the authors followed an idea to decrease the amount of parallel executions by serializing them as much as possible. Thus, the dependencies between the subtasks of a DAG task are replaced by intermediate offsets and deadlines. This is done for

the sake of transforming the DAG task to a set of independent constrained deadline-sequential tasks with the corresponding timing attributes. The researchers proposed to construct a master thread by stretching the critical path of the DAG until the deadline of the task. In the case that there would be more than a single critical path, only one of them is chosen arbitrarily, and then it is executed by a single processor. The subtasks that do not belong to this path are executed by the independent parallel threads on the rest of the processors. In particular, the authors applied the Global EDF algorithm to shedule this workload and derived a resource augmentation bound. Nevertheless, the stretching algorithm is considered to be a preparatory phase for the scheduling process. Thus, it can be combined with other scheduling algorithms as well.

All the works on DAG tasks discussed so far implicitly share a common assumption, that every time when an instance of a DAG task is released, all the subtasks of a respective DAG will be eventually activated. Therefore, the control flow information is not presented in the DAG. Fonseca et al. proposed [64] an approach to incorporate such information into a DAG task model. The authors represented a control-flow powered DAG task as a collection of the ordinary DAGs, each one providing a different execution flow. The authors proposed an algorithm for combining these DAGs into a synchronous parallel task preserving the original timing attributes and precedence constraints. This approach benefits from a potential for utilizing existing approaches for DAG schedulability analysis. Unfortunately, the approach does not scale with respect to the number of subtasks, given that the number of possible flows has a potential for drastic increase.

Melani et al. [144] and Baruah et al. [17] addressed the scenario of control-flow divergence in DAGs from a different angle. The authors introduced to a DAG task model a notion of *conditional vertex pair* – a pair of vertexes representing a conditional segment of a code . The first vertex in the pair represents a point in the code where the

conditional statement has to be evaluated and the control-flow diverges into several possible branches. The second vertex represents a point of convergence where all the respective branches have to meet. Except these two vertexes there should be no other vertex that the branches can share. Moreover, there should not be any edge that connects a vertex of a branch to the vertex in the DAG that does not belong to that branch. The total number of the alternative branches which correspond to a single pair of conditional vertexes is referred as a *branching factor*. The authors assumed that if the control-flow passed through the first vertex of the conditional pair, it has to reach the second vertex of the pair. Hence, this assumptions would require additional programming model restrictions or coding conventions when when applying the conditional DAG model to the state-of-the-art parallel programming frameworks, e.g., OpenMP, CilkPlus, TBB, etc.

The researchers considered a problem of the global scheduling of parallel tasks on a platform composed of identical processors. The tasks are characterized by a sporadic arrival pattern and constrained relative deadlines. Every task is expressed as a DAG supporting conditional vertexes. Each vertex within every DAG is characterized by a specific worst-case execution requirement.

Melani et al. introduced [144] the notion of the *worst-case workload* of a conditional DAG-task – the maximum (among all possible conditional branches) time required to execute that task on a dedicated uniprocessor. The authors used this notion to generalize a parameter to the non-conditional DAGs, called *volume* – the sum of the WCETs of all the vertexes. This generalization allowed them to reduce the level of pessimism, since in the general case not all the subtasks are necessarily required to execute. The researchers used the worst-case workload to find the upper-bounds on inter-task and intra-task interfereces and presented an algorithm of polynomial complexity to compute this parameter. Based on it, a sufficient pseudo-polynomial schedulability test was derived.

Baruah et al. [17] proposed to transform conditional sporadic DAG tasks to non-conditional sporadic DAG tasks. The approach is based on the notion of *work* – the amount of execution that can be generated by a collection of jobs of the task under consideration. The authors utilized the concept of the *work function* [32] that returns the amount of work generated during the time interval of a specific length. The researchers presented a proof that states that for any conditional sporadic DAG task there exists a non-conditional sporadic DAG task with an equal work function.

The authors proposed an iterative transformation algorithm, where each iteration can be briefly described as follows:

1. Identifying an *inner-most conditional construct* – the one that does not contain any nested conditional constructs.

2. Creating a non-conditional construct "equivalent" to the one identified during the previous step.

3. Replacing the construct identified on the first step with the construct created on the second step.

The non-conditional analog and the original conditional construct should be characterized by equal values of the following parameters: the work function, the length of the longest path, the workload, the relative deadline and the minimum inter-arrival time.

This transformation strategy is exploited for the sake of applying the schedulability analysis of non-conditional DAGs [32], and then to draw a conclusion about the original conditional DAG. The authors presented a proof stating that the conditional task set is schedulable by global EDF if and only if the corresponding non-conditional task set is schedulable by the corresponding algorithm.

Li et al. presented [122] a generalization of the concept of partitioned scheduling in the context of constrained-deadline parallel sporadic DAG tasks called *federated*

scheduling. The authors considered a task to be *heavy* if its utilization exceeds 100%. In federated, scheduling each heavy task gets an exclusive access to a set of processors. Hence, all the processors available in a system are partitioned into $n + 1$ "clusters" where n is the number of heavy tasks. An extra cluster is dedicated to the *light* tasks – those, whose utilization is less than 100%. Although the light tasks in this shared cluster run sequentially, the authors allowed these tasks to execute on more than a single processor. Thus, the cluster for light tasks does not gain the benefits of simpler analysis and implementation traditionally afforded by the partitioned approach. The researchers presented a proof of a capacity augmentation bound for this model.

Baruah introduced [15] an analysis of federated scheduling for the case of arbitrary-deadline sporadic parallel DAG tasks and presented a proof that by moving to a more general deadline-model there is no loss in terms of speed-up metric. Baruah extended [16] the definition of federated scheduling by adding a restriction that the light tasks in the respective cluster are partitioned among the respective processors at DAG granularity. Therefore, light tasks cannot migrate.

The approaches discussed above, as the schedulability analysis in general, rely on the task attributes to be input by the designer. It is the Worst-Case Execution Time analysis (WCET) analysis which provides the C_i for each task.

3.5 Worst-case execution time analysis

Solid timing analysis is a principal stage in real-time systems design. It is needed to ensure that the tasks meet their timing requirements and the interrupt latencies are bounded by respective limits. For hard real-time systems obtaining the worst-case execution time (WCET) of each real-time task is of paramount importance. In some cases, for a single task it is possible to obtain the worst-case input, but a general approach to finding WCET is through using WCET analysis. Developing such an analysis is subject to some fundamental hurdles. Namely, developing precise and

accurate model of hardware execution latency (i), determining the timing behaviour of the task which depends on the history of previously executed instructions that exerted influence on the hardware state (ii), developing approaches for characterization of the worst-case temporal behaviour and proving that it has specific mathematical properties to ascertain confidence bounds for the real-time system (iii). All of these challenges are subject to extensive research efforts led by academia and industry, to be described in the following sections.

3.5.1 Sources of performance and unpredictability

Originated from one-off government-funded developments characterized by enormous cost, embedded systems emphatically move towards cheaper commercial off-the-shelf components. As a well-known acknowledgement of such a trend could be a usage of general purpose processors in real-time industry domain. Being attractive not only by price but also by rapidly growing computational capabilities, low power consumption and compact form factor, mass product processors have a tough drawback for real-time embedded system designers. Strong focus on average case performance, that naturally is the most important concern for general-purpose hardware, makes it very unpredictable from the real-time viewpoint. Therefore, here we would like to highlight hardware optimizations that are known to cause problem of unpredictability of such processors.

Today, memory technology does not seem to be able to catch up with processor speed. Therefore, in modern computer systems, there is a huge gap between processor clocks and memory access times also known as *memory gap*. When the processor has to access off-chip memory it means that the data will come tens, hundreds (or even more) clock cycles later, hence the processor needs to wait until it will be able to resume processing. Such scenarios are usually referred as *wait states* and cause waste of performance and energy. They may even lead to unnecessary overheating

if no energy conservation strategy is applied to prevent the processor from spinning uselessly waiting for data. To diminish negative effects of wait states several processor hardware optimization techniques were introduced, namely: caching, instruction pipelining, instructions prefetching, branch prediction, simultaneous multithreading. All together, these techniques show tremendous achievements in reducing the problem in the average case, however from the real-time system timing analysis viewpoint, they make computer hardware too complex, much less deterministic and, therefore, poorly suitable for the worst-case timing analysis.

Probably, one of the most natural solution to the problem of reducing the number of wait states should be diminishing the off-chip data traffic. Hence, hardware designers do much effort to keep data as close to the processor as possible. Unfortunately, making low-latency memory for processors that run at high clock rate is a big challenge, therefore, chip manufacturers opt for a design that includes multiple levels of memory subsystem to combine cost and performance. Ordering these memory levels from the smaller but faster to the bigger but slower, the resulting sequence is the following: register file, cache memory, main memory.

Very common principle of processor design that is known to be dramatically beneficial for average-case workload, is to exploit execution history of the program based on temporal and spatial locality. This principle is the essence of an idea behind the cache memory which on one hand has significantly more storage comparing to registers, and on the other hand, is much tiny and faster accessible comparing to the main memory.

Caches substantially improve the average application performance reducing data access time of a general-purpose processor by one or even two orders of magnitude. However, for the timing analysis cache memory introduces a tremendous complexity: the execution time of the program becomes extremely dependent on the execution history. Even assuming that cache characteristics like size, associativity, etc., would

be documented by the chip-makers in every detail, which is often far from being the case, potential option to model cache behavior in software do not help much in real-time systems perspective. Unfortunately, cache simulation usually does not provide safe enough results to determine worst-case timing behavior. The reason is that claiming simulation to be safe, one needs to cover all possible program paths which would require an exponential number of input data to be analyzed and in most of the cases is not tractable subject to the problem size and computational capacities available.

Many works [182], [163], [87], [172], [123], [160] on WCET analysis have been done for single-core processor, however, due to high complexity features of general-purpose CPUs namely pipelines, caches, branch prediction, speculative execution and out-of-order execution, still it is very hard to obtain accurate WCET [24].

For preemptive multitasking environment cache analysis is way more difficult, given that cache state depends on the history of the execution. The essence of the problem is that any cache line potentially can be influenced by an instruction which is placed in the same piece of code or another module or even another program. This challenge of inter-task cache interference was approached by "footprint-based" methods [19], [35], [116], where task footprint is a part of cache memory used by a corresponding task. Footprints overlapping demonstrates how tasks compete for the same cache area, and therefore, determinates cache related task switching overhead. For example, Busquets *et al.* [35] incorporated instruction cache support into fixed priority schedulability analysis. In their simulations the authors compared cached Response Time Schedulability Analysis with cache partitioning (single cache partition per task) and with cached Rate Monotonic Schedulability Analysis. The authors assumed a processor with one level on-chip instruction cache (no data cache present) that is entirely refilled every time when task context switch occurs and therefore experience cold start. The authors consider such refilling feature of their processor

model as a pessimistic assumption, however, this is true only in case we consider cache as an isolated resource disregarding the big picture of the system. As it was later shown by Lundqvist *et al.* [139], such local worst-case assumptions lead to a common pitfall. The reason is, modern processors include multiple factors that influence each other at runtime.

Let us consider pipelines that accelerate processing by overlapping execution if there are enough instructions to keep the pipeline full. Choosing the pipeline state that will cause the longest overall execution time of the program would be impossible without knowing the complete instruction sequence. Another aspect of the problem is that all state-of-art processors are dynamically scheduled – the instructions could be executed out-of-program-order. With the help of simplified model, inspired by PowerPC processor, that includes integer unit (for ADD instruction), multiple cycle integer unit (for MUL instruction), load/store unit – all supported by reservation stations, Lundqvist *et al.* considered series of possible scheduling scenarios. Taking into account instruction sequence, dependency between instructions, instruction dispatching time, the authors demonstrated [139] how out-of-order scheduling of arithmetic instructions may lead to timing anomalies – scenarios where the local worst-case hardware behavior does not lead to the longest overall program timing. Particularly, simply assuming a cache miss in a dynamically scheduled processor is not safe enough given that in some cases it may result in shorter overall timing comparing to cache hit scenario. The authors presented code modification techniques for restricting out-of-order execution, however, their approach would become practical only in case of architectural support for cache states control. Also, it prohibits fully preemptive scheduling and similar to cache there is a need of explicit control on pipeline stages. Given that the model of the processor was greatly simplified, for more realistic hardware model there is a need of accounting on the effects of the following factors that are usually considered as primary sources of unpredictability

and performance. Those are: speculative execution (performing instructions before being sure if the result will be needed); branch prediction (proceeding with further instructions without waiting for the result of a branch); prefetching (getting data earlier then there is a demand); out-of-order scheduling of load/store instructions; memory contention between instruction and data.

Wilhelm *et al.* showed [183] how detrimental such hardware optimizations could be for predictability of real-time system.

Methods of WCET analysis are divided in two classes: *static methods* and *measurement based methods*.

3.5.2 Static methods

Methods of static WCET analysis use abstractions to cover possible paths of execution and processor behavior at the cost of the need to create processor-specific models. Static methods do not require running the program under analysis, and therefore, complex and expensive equipment to simulate the target computer system is not needed. However, static analysis methods require a deep knowledge of the target hardware as well as and the ability to reason about its state. Unfortunately, an increase in complexity at hardware and software levels of conventional real-time embedded systems makes these methods very challenging in terms of tractability, even subject to an assumption that all the hardware features are documented in every detail. This is due to an extreme number of possible states of the state-of-the-art hardware which leads to a combinatorial explosion when enumerating those even for a relatively simple code snippet. Thus, given that current static methods do not scale up to the increase of complexity, alternative approaches are needed to keep pace with the hardware evolving fast.

Another common problem of static WCET analysis is overestimating the real value of the WCET drastically. The overestimation of WCET can be tolerated by

decreasing the degree of safety of the estimate. The corresponding methods use program execution measurements instead of analyzing the processor behavior subject to the processor model (like the static methods do) and therefore, are classified as measurement-based.

3.5.3 Measurement-based methods

Unlike static program analysis which is used to obtain dynamic context and characteristics of the computational entity without its execution, measurements require that the respective program code should be run either on the hardware or on its software simulator. Since, in most of the cases, running experiments for all possible control flows (code paths and inputs) is intractable, measurements can only demonstrate typical dynamic behavior of the program, rather than the execution time bounds.

Even though being potentially unsafe, measurement-based techniques are widely accepted in the industry thanks to their practical feasibility. The general scheme of such industrial techniques includes [182] three steps:

- preparing high-coverage input data;
- conducting extensive experiments on the program initialized with that data and recording the longest execution time that was observed (high watermark execution time);
- adding a constant (called an *engineering margin*) to the value of the longest execution time, subject to the assumption that the margin is big enough to cover any unanticipated worst-case timing scenario of real-time system behavior.

Although these end-to-end measurements are used in the industry, the produced results may underestimate the real worst case. However, they give an idea about the execution in the average case and the probability of the worst-case occurrence. An attempt to replace empirical techniques of determining engineering margin with

scientific approaches based on a proper theoretical background, led to multiple efforts in applying results that have been developed by the probabilistic and statistic research communities to real-time systems domain.

3.6 Probabilistic real-time systems

The idea behind probabilistic analysis is to estimate the chances of the scenarios in the future based on some model of probability. For example, when tossing a coin one can consider that the probability of it falling on a head or a tail is equal to $\frac{1}{2}$ subject to the assumptions that the coin is "fair", it will not disappear before falling, etc. The use of probabilistic approaches in real-time systems in the context of timing analysis is based on an idea that an exceedance of the WCET of the software can be modeled as a failure of the system. In this sense, the mechanical parts, electrical hardware and eventually a software have a common aspect in their reliability behavior – all of them have some probability of failure.

The application of the probabilistic analysis to real-time systems is done through the introduction of some system parameters characterized by random behavior. In the context of our research, we are interested in works that extend this approach to the timing of the execution. Thus, we briefly present the works that deal with the randomized worst-case execution requirements.

3.6.1 Probabilistic response time analysis

Liu et al. showed [126] that in the context of a set of independent preemptive periodic tasks with fixed priority, the most unfavorable scenario for uniprocessor scheduling of a particular task occurs when it releases its task-instance together with all the other tasks of an equal or higher priority at the same time instant (also known as *critical time instant*). In other words, to bound the worst-case response time of a particular task, one should look at the scenario, when that task releases its job at the critical

instant. Based on this observation, Lehoczky et al. introduced [117] the concept of *time demand function* – a function of time which returns the cumulative demand on the processor resource for the time period started from the critical instance. Based on this concept the authors presented an exact characterization of the Rate Monotonic algorithm [126] in terms of its scheduling ability. Moreover, the researchers applied the probabilistic analysis to characterize a corresponding scheduling bound in the average case. The authors considered randomly generated task sets by introducing the cumulative distribution functions for both the periods and the execution requirements of the tasks. Their simulations demonstrated the gap between the worst case and the average case in the context of the uniform distribution.

Tia et al. based their work [176] on the Time Demand Analysis [117] discussed above. The authors considered a constrained-deadline task system with periodic releases, variable execution times and fixed priorities executing on a uniprocessor computer system. The authors modeled the execution times of the tasks as random variables and presented an analysis that derives the probability of missing the deadline for every task. For a task under consideration, the analysis provides a bound on the total amount of the execution time of the higher priority tasks. The respective *probabilistic time-demand method* is an extension of an exact Rate Monotonic schedulability test [117]. Assuming that the distributions of the execution times of the tasks are known, the researchers substituted sums of the execution requirements with the *convolutions* – mathematical operations that return the area overlap of the input probability distribution functions. Thus, for every task, the probability distribution function of the corresponding response time is derived, assuming the worst-case release time. The proposed algorithm tractably performs convolutions for at most ten higher priority tasks. In case there is a need to account for the effects of more tasks, it uses the Central Limit Theorem [167] for the sake of approximation, abstracting away from the respective distributions.

Lehoczky observed [119] that the classical definition [126] of a real-time task set is not well suited for some of the application domains that require timing guarantees. Specifically, the author pointed out that the basic assumption about the static task set and the deterministic worst-case execution requirements do not fit the applications of real-time communication on automatic teller machine networks and multimedia processing. The researcher proposed to express randomness in job releases and their execution times with the help of queueing theory, considering a real-time *job* as a *customer* in the queueing terminology. The model is based on the following assumptions:

- There is a queue of customers waiting for a single processor operating at a specified rate;
- The arrival of the customers is described as a Poisson process with a specified rate;
- The mean of the execution requirements required for processing the customers is given;
- Each customer has a variable relative deadline described by a given cumulative distribution function;
- The customers in a queue are processed according to a defined *queueing discipline*;
- The processing of the customer can be preempted at zero cost.

Since the classical queueing theory does not have a concept to be used for expressing the execution requirement of every particular job, extending the model to account for execution requirements is complicated. The difficulty appears because the *state variables* of an extended model get an unbounded dimension. Therefore, the author proposed to add a *heavy traffic* assumption – the average utilization of the processor

has to be almost full. This allows to apply additional probabilistic methodology that simplifies the analysis of the model. However, for the scenarios when the system does not experience the heavy workload, the model becomes too pessimistic. Nevertheless, for such scenarios, the author suggested to use the model for deriving the worst-case bound on system performance. Another source of possible shortcoming of the model is that all the customers are forced to have equal probabilistic characteristics(as stated in the assumptions above) which would not necessarily be the case in a real-life workload.

Atlas et al. pointed out [4] to the soft real-time applications where the miss of the deadline is acceptable subject to the requirement that for the overall system the number of the deadlines met is higher than some threshold specified by the designers. As the work discussed above, the researchers also considered preemptive Rate Monotonic scheduling of periodic tasks with variable execution times. Similarly to the classical Rate Monotonic analysis [126], this approach also consists of a feasibility test and a scheduling algorithm. Both are linked to the concept of the quality-of-service. The authors defined the quality-of-service of the task as a probability of a job (selected randomly) of that task meeting its deadline, considering an infinitely long history of system operation. The feasibility test ensures whether for a given task set it is possible to satisfy its quality-of-service requirements. The proposed scheduling algorithm consists of an *admission controller* and a scheduler. At the release of every job, the admission controller has to admit or reject this job to be considered for scheduling. This decision is taken based on the likelihood of the job to meet its deadline. The admitted jobs are scheduled according to the respective priorities.

Gardner et al. presented an analysis based on General Time Demand Analysis of Lehoczky et al. [118]. Their Stochastic Time Demand Analysis [67] derives the lower bound on the probability that the jobs of the task system under consideration will meet their deadlines. Similarly to Probabilistic Time Demand Analysis [176] of Tia

et al., the proposed analysis makes a simplifying assumption that every particular task releases its instance together with all the higher priority tasks. This critical instant assumption allowed the authors to compute an upper bound on the probability of deadline miss. Gardner et al. succeeded in relaxing the constrained-deadline assumption made by Tia et al. Another novel aspect presented is that the researchers considered the time demand from the perspective of the task instance being analyzed, rather than the entire busy period. Thus, the probability that the job meets its deadline is equal to the probability of the time interval between its release and its deadline being sufficient to meet the time demand of the system. However, both Stochastic Time Demand Analysis [67] and Probabilistic Time Demand Analysis [176] focus on uniprocessors.

Bernat et al. introduced [26] the concept of an *execution profile* – a characterization of a code segment using the frequencies of the occurrences of possible events caused by that code. Although an idea behind the execution profile can be applied to characterize different aspects of the computer system‘s behavior (e.g., possible memory layout, cache accesses, etc.), the authors concentrated on the *execution time profiles* (ETPs). The authors proposed to define an execution time of a code segment with the help of marking some stage of the instruction pipeline of the processor under consideration to be a point of reference. Then, the *execution time* of a code segment is defined as a time interval from the moment when the first instruction of that segment enters the marked pipeline stage, until the moment when the last instruction of the segment leaves that stage. Assuming that the frequencies of possible execution times of the code segment under consideration are collected (e.g., using analytical methods, simulations or measurements), the researchers represented the respective execution time profile using a probability distribution. The authors proposed mathematical representations of the cumulative execution time profiles for different scenarios of the code segment executions: conditional statements, loops, sequential execution. For

the latter, the researchers considered the following three cases:

- the code segments are independent;
- the code segments are dependent, and the dependency information is known;
- the code segments are dependent, but the dependency information is not known.

Based on the concept of convolution, for each of the cases presented above, the authors presented a separate operation for combining execution time profiles of the code segments. Thus, the ETPs of the code segments that belong to the same execution path can be combined to the ETP of a whole path, and therefore, the model of the longest execution of the code is derived. The usefulness of the algebra of ETPs developed by the researchers is also motivated by an idea of combining the execution time profiles with the analytical methods for deriving the worst-case program paths. This idea made it possible to develop methodologies that utilize the techniques from static, measurement-based and hybrid timing analyses. For example, the path derived with the help of the deterministic control-flow analysis can be augmented with the execution time profiles obtained by using measurements.

Such techniques, combining a probabilistic view on system‘s behavior with well-established timing analysis approaches, form *Probabilistic Timing Analysis* (PTA).

3.6.2 Probabilistic timing analysis

The objective of this analysis is to provide such bounds on the execution time behavior of the system‘s software that the probability of timing failure of the whole real-time system will be kept below the acceptance threshold specified by certification or quality-of-service requirements. The worst-case timing behavior of the software is no longer considered as a single value of an execution time, as it is done in the traditional WCET analysis approaches. The PTA introduces the notion of probabilistic Worst-Case Execution Time (pWCET) – the probability distribution of possible execution

times such that it bounds the probability that the execution time may exceed a value (timing) given in a distribution. Therefore, the analysis is mainly utilized to obtain the timing that has a probability of occurrence that is less than or equal to the acceptance threshold.

Based on how the probabilities of possible execution times are derived, two branches of PTA are under active development:

- Static Probabilistic Timing Analysis (SPTA) – the a-priori probabilities are derived statically from the model of the system;
- Measurement-Based Probabilistic Timing Analysis (MBPTA) – the probabilities are derived a-posteriori from the end-to-end runs.

3.6.2.1 Static Probabilistic Timing Analysis Cazorla et al. proposed [37] Static Probabilistic Timing Analysis whose principal stages can be briefly outlined in the following way:

- Obtaining a-priori probabilities for the timings of the execution entities (e.g., individual instructions or code segments);
- Deriving discrete probability distributions for the respective execution entities;
- Combining these distributions into a single one, subject to a given worst-case sequence of the execution entities and an assumption that the execution times of the entities are statistically independent;
- Applying Extreme Value Theory assuming that the input data are independent and identically distributed.

The assumption about statistically independent execution entities does not fit the vast majority of computing systems available, therefore, the authors presented a different paradigm. The researchers proposed to tackle the problem of non-deterministic

timing behavior of modern computing systems by getting rid of the execution history in such systems. The authors suggested to introduce the randomization in the timing behavior of such systems being the candidates for an adoption in the real-time domain. In other words, the timing of an arbitrary instruction under consideration must be independent of the previous executions in the computing system, even though there might be a logical dependence between the current instruction and some instructions executed previously.

Considering which components of a computing system should (or need not) be randomized, the researchers presented the following classification:

1. Fixed-latency components;
2. Components that have latencies with low variability;
3. Predictable components that have significant variability;
4. Unpredictable (or intractably predictable) components that have significantly variable latency.

The analysis of the components that belong to the first class can be done in a straightforward way by accounting for the fixed latency. For the second class of components which are characterized by a small difference between the worst-case latency and the best-case latency, a reasonable way would be to account always for the worst-case. This would be acceptable subject to an assumption that the pessimism added by such a simple approach is negligible when compared to the overall worst-case execution requirement. The definition of the third class of components imposes that there exist tractable methods that are able to accurately predict the respective latency. While the analysis of the components that belong to any one from the first three classes can be done by performing affordable cost/effort activities, the fourth class is the one that the authors suggest to be a right fit for the randomization idea.

The researchers argued that the typical example of such components is a cache memory, which inherently depends on the execution history. Instead of using a deterministic replacement policy, the authors proposed to randomly select the cache line to be evicted from the cache set. Hence, the eviction of the cache line under consideration becomes independent of previous accesses. This random replacement policy brought the authors to the idea to characterize the behavior of the cache probabilistically based on the *reuse distance* of the memory location under consideration, that shows how recent the previous access to this location was. More formally, the reuse distance is the number of accesses to distinct memory addresses that have happened between the two most recent accesses to the memory location of interest. For the fully-associative caches with random replacement policy, the use of the reuse distances liberates the analysis from the dependency on the memory layout, since the data from the memory location can be put into an arbitrary cache line.

However, in case of set-associative or direct-mapped caches, the address of the respective memory location determines the set, in which the data will be stored. Therefore, for such cases the researchers proposed to randomize both, the placement and the replacement policies. The authors pointed that the randomization of the placement policy does not need to be explicit. Given that the placement directly depends on the mapping of the data in memory, the randomization can be implicitly implemented on the side of runtime systems by doing random memory allocation (e.g., the memory manager presented [25] by Berger et al.).

Cazorla et al. considered [37] a theoretical model of the computing system powered by a CPU with a pipeline, an instruction cache and a data cache. The in-order pipeline under consideration is characterized by a-priori known latencies of the instructions. These latencies are fixed for all types of instructions except the memory accesses. The latencies of the memory instructions depend on the data cache subject to an assumption about permanent hits to the instruction cache. Thus, the latency of a

memory instruction has two possible values, corresponding to the data-cache hit and to the data-cache miss.

Therefore, the computing system under consideration includes a single source of timing variability – a data cache. This cache is supposed to be fully-associative and every access to it causes an eviction of a random cache line from the cache. Thus, this *evict-on-access* replacement policy is enough to make the cache under consideration be time-randomized.

The researchers assumed programs with a single possible path and a fixed input data. Thanks to these assumptions, there was no need to perform path analysis and loop bound analysis. The authors argued that the benefit of their static probabilistic timing analysis is based on the use of the reuse distances, since there is no need to have runtime information about how the data is mapped to the memory as it is required in traditional static timing analysis approaches.

In a probabilistic timing analysis presented by the authors, the probability distributions of the individual instructions are determined statically from the model of the computing system discussed above. The model of the randomized cache allowed the researchers to consider these individual instructions as *independent* random variables. This is an implementation of a principle of time-randomized hardware design promoted by this work. Hence, the convolution operation could be applied to combine the respective discrete distributions into a single distribution of the execution time of a sequence of instructions. The further analysis is based on the respective inverse cumulative distribution function also known as the *exceedance function*. It is analyzed for which estimation of the worst-case execution requirements the value of the exceedance function falls below the required acceptance threshold. Hence, the respective execution time is considered to be not exceeded with a given level of confidence.

In terms of the experiments, the computing system discussed above was simulated

to process a synthetic benchmark characterized by a single control path consisting of a fixed number of distinct memory accesses.

The authors compared the models of time-randomized cache and a few configurations of LRU caches, in the contexts when some of the memory accesses are unknown and consequently the LRU cache has to be flushed. These experiments showed how the estimation of the worst-case execution requirement increases with the increasing of the number of the unknown accesses.

The further experiments demonstrated the sensitivity of the time-randomized cache to the size of the cache and to the acceptance threshold for the scenarios with different amount of unknown memory accesses.

The work discussed above estimates the pWCET of the entity of execution (e.g., task) that runs in isolation. However, the more realistic scenario should account for preemptions, when the execution of the instructions that belong to different tasks may be interleaved on the processor. In such cases, the instructions of the preempting task affect the context of the execution of the preempted task, for instance the probabilities of the required data being in cache. The effect of cache misses affecting the worst-case execution requirement is usually referred to as Cache Related Preemption Delay (CRPD) that is applicable to the computing systems running under preemptive scheduling.

Davis et al. extended [50] the static probabilistic timing analysis presented [37] by Cazorla et al. to account on probabilistic Cache Related Preemption Delay (pCRPD). Their analysis provided an upper bound on the inverse cumulative distribution function of pWCET for the case of the following computing system.

The authors considered a uniprocessor system, however, in terms of caches they assume an instruction cache only. The instructions executed by the processor are stored in *memory blocks*, such that multiple instructions may belong to the same block. The researchers assumed that the instructions are characterized by two fixed

values: an execution time in case of cache-hit and in case of cache-miss. Moreover, to simplify the analysis the authors assumed that all the instructions share the same cache-hit latencies and the same cache-miss latencies.

The authors considered *evict-on-miss* random replacement policy, according to which every time a cache-miss occurs, some randomly selected cache line gets evicted, while the block (that contains the instruction under consideration) fetched from the memory is loaded into the instruction cache.

In their experiments researchers simulated the given computer system running a suite of real-time benchmarks [81] that contains both single-path and multi-path programs. The authors compared the performances of the systems powered with the evict-on-miss and the evict-on-access instruction caches and drew the conclusion that the evict-on-miss is more suitable.

Similarly to the work of Cazorla et al. [37], this approach is based on the reuse distances. Davis et al. assumed that a deterministic reuse distance for each instruction is given as an input to the analysis. The authors argued that this aspect provides a significant margin of safety when compared to the option of using probabilistic reuse distances.

The works discussed above implicitly shared a common assumption: all of them assume hardware that is functioning correctly. However, a fine-grained fabrication of the state-of-the-art integrated circuits imposes a drastic increase in probability of failure among the corresponding silicon primitives [147]. Motivated by the reasoning that such probabilities increase exponentially with the decrease of the distance between the transistors, Hardy at al. addressed [85] this phenomena in the context of static probabilistic timing analysis research.

The authors considered a uniprocessor computing system equipped with a single-level instruction cache subject to an assumption about the absence of timing anomalies. Thus, the worst-case behavior is assumed to be imposed by the cache misses.

The cache is characterized by an LRU replacement policy and some level of non-determinism that comes from its fault bits. The researchers assumed such bits to be free from the effects of *transient faults*, thus, if the bit fails, this fault is *permanent*.

The instruction cache is assumed to be the only component of the computing system that may experience failures which are detected using post-manufacturing tests, Error Control Correction, etc. While the LRU-stack bits are assumed to be reliable, the ordinary bits are considered to have an equal probability of failure, which is supposed to be given as an input to the analysis. If a bit experiences a fault, the corresponding cache block is marked as *disabled* and the size of the cache is reduced. The decrease of the cache capacity leads to additional *fault-induced* cache misses which are considered in the analysis.

The approach proposed by the researchers is partly based on static timing analysis, particularly, path and cache analyses that are combined with the analysis of probabilistic aspect of the computing system that is imposed by the instruction cache. In terms of low-level analysis, subject to the given cache configuration, the worst-case behavior of each memory reference is classified using the approach presented [175] by Theiling et al. On the other hand, in terms of high-level analysis, from a given program, the worst-case execution path is derived. Then, an upper bound on the WCET is computed using Implicit Path Enumeration Technique [182] that is based on Integer Linear Programming. Then, the obtained knowledge is augmented by the probabilistic analysis that for a given probability of bit failure, evaluates the additional fault-induced cache misses that may happen. Based on the distribution of the latencies that occurred because of such faults, the overall pWCET can be obtained, that is supposed to be used to ensure that the timing requirements to the computing system under consideration are met.

In terms of the experiments, the authors also developed a brute-force method that for a relatively small problem instant performs an exhaustive enumeration of all

potential fault bits. This method that gives the worst-case timing penalty imposed by the instruction cache failures is used to demonstrate the quality of the probabilistic analysis discussed above.

The work [37] by Cazorla et al., that we were discussing at the beginning of this subsection, also provided an intuition for another variant of PTA, Measurement-Based Probabilistic Timing Analysis.

3.6.2.2 Measurement-Based Probabilistic Timing Analysis MBPTA is based on the traces of the analyzed program running on the target platform. Thus, it has a very strong practical benefit, especially, in the scenario when the details of the hardware/software organization of the system are kept secret (which is often the case). However, in terms of timing profile, the execution time measurements provide not the probability distribution, but only the *frequency* distribution observed within a finite interval of time during which the experiments were conducted. This fact provides a serious limitation for the systems with a low acceptance threshold that would require high precision on the assigned probabilities, and therefore, an extremely large number of traces.

The considerations presented above have a paramount importance because of the following reasons:

- Providing a huge number of traces might not be feasible for many real-life applications;
- The occurrence of the worst case can be considered as a "rare event".

Even though the rare events can be considered as "improbable" they are also characterized by their drastic impacts, and hence, attract serious attention by the statistic research community. Rare event theories focus on the *tails* of probability distributions that correspond to "low probabilities", to analyze how the random variable under consideration deviates from its expected value.

Cazorla et al. considered [37] two rare event theories to be applicable to the problem of estimating pWCET: Large Deviations Theory (LDT) [177] and Extreme Value Theory (EVT) [70].

Originally, these theories extend the Law of Large Numbers and the Central Limit Theorem [167]. According to these results, for a sample that contains a sufficiently large number of observations of a random variable under consideration:

- The arithmetic mean of the observed values converges to the respective expected value;
- That arithmetic mean is approximately normally distributed;
- The (*tail*) probability of that arithmetic mean being greater than a given value (specified beforehand) can be approximated.

Unfortunately, the latter approximation derived using the Central Limit Theorem might not be accurate for the cases when the value specified for the computation of the tail probability is relatively far from the expectation value of the corresponding variable. Another drawback is known when the number of observations grows to the infinity, since the Central Limit Theorem does not provide the information about the convergence of the tail probability.

Large Deviations Theory tackles these problems by focusing on tail probabilities, however, LDT analyzes the sum of random variables. Thus, Cazorla et al. argued [37] that it could be potentially applied to the combination of execution traces (e.g., each trace for a different module of the software).

For a single trace represented by many measurements, an established idea is to apply Extreme Value Theory. EVT is a branch of statistics that for a large enough sample of a random variable, estimates the probability of exceeding *all* its values observed previously. In other words, this discipline studies the extreme deviations from the median of the probability distribution of the random variable under consideration.

The traditional EVT is based on two assumptions about the observations:

- The observations are independent in a sense that the outcome of the observation under consideration is not correlated with an outcome of any other observation that has already happened.
- The observations are identically distributed meaning that the probability of the observation under consideration is identical to the probability of the same observation but from another sample.

Therefore, an application of such flavor of EVT to the traces of measurements requires these execution traces to be mutually independent and to have the same probability distribution function.

Edgar et al. proposed [53] to estimate the worst-case execution time of the real-time tasks using EVT. Based on the sampled execution timings, the authors computed the *scale* and the *location* parameters for deriving the probability distribution function. The researchers opted for the Gumbel distribution [80] from the family of continuous probability distributions, also known as Generalized Extreme Value (GEV) distribution. However, contrary to EVT that considers [21] only those random values that are maximum or minimum from sufficiently big sets of other random values, the researchers fit all the observed measurements to the Gumbel distribution.

Hansen et al. improved [84] the approach discussed above by suggesting to analyze only maximum values derived according to some principle from a given sample of the measured execution times. First, the authors grouped the measurements into blocks of an equal length. Then, from each block the maximum value was taken for construction of a new set of "block maximum" values. Thus, instead of fitting the raw execution timings to the Gumbel distribution, the researchers used the *Block Maxima* method [21] that provided the maximum random values.

Griffin et al. noticed [79] that in computing systems the notion of time is discrete,

therefore, a program, as an object of timing analysis, cannot terminate at an arbitrary point of some continuous time interval. This is an important observation in the context of previously discussed timing analysis works that apply traditional EVT to the estimation of the worst-case execution requirement. The authors warned that fitting a continuous distribution (e.g., Gumbel) to discrete execution times might be unsafe in cases when the distribution function would not bound the observed execution time from above. For such cases, the researchers proposed to use the following options:

1: Add a safe offset to the distribution;

2: Overestimate the respective discrete exceedance distribution function by fitting its upper bound by a continuous function.

The first option is applicable only for the scenarios when such an offset can be safely derived. The second option is more realistic, however, it is likely to add a significant pessimism.

Cazorla et al. argued [37] that alternatively to the Block Maxima method [21], the unsafeness from applying EVT to a set of discrete values can be eliminated by using Peaks-Over-Threshold method. Unlike Block Maxima, this method does not compare the value under consideration against the extreme value within the block, but against a specified threshold. Thus, only the values that are more extreme than the threshold will be taken into account.

Block Maxima and Peaks-Over-Threshold help EVT to provide bounds on the probabilistic worst-case execution requirement, however, the difficulty of their proper usage comes from the determination of the block size and the value of the threshold respectively. The choice on the values of these parameters determines the portion of the original distribution that will be considered when fitting the continuous GEV distribution and is usually a result of significant empirical efforts.

Griffin et al. also pointed out [79] that the assumption about the independence

of the observations is a limitation of such works that apply traditional EVT to the estimation of the worst-case execution requirement. The authors argued that the real-life computing systems do not satisfy this assumption. Especially, this is true for the hardware that was designed with the focus on optimizing the average-case performance.

To satisfy the independence requirement, Lu et al. proposed [138] to transform the original sequence of observations using sampling techniques. The output of their sampling mechanism can be an input to the EVT without raising a problem of dependent observations. However, this approach does not provide a guarantee that the resulting sequence of observations has the same statistical properties as the original one.

Cucu-Grosjean et al. addressed [48] the independence requirement by relying on the model of the computing system powered with time-randomized hardware. In this sense, the authors extended the approach presented by Cazorla et al. [37] who suggested to upper-bound the timing behavior of the hardware when that allows an acceptable level of pessimism and to randomize those hardware resources whose upper-bounds on the worst-case timings would be too high. Cucu-Grosjean et al. provided modeling of fully-associative data and instruction caches featuring a random-replacement policy. The time randomized hardware allowed the authors to satisfy the independence hypothesis required for the application of the traditional EVT.

The researchers assumed that the latencies of the processor pipeline stages in the instruction cycle are fixed except the fetch stage, which is characterized by only two possible options for the case of cache hit and the case of cache miss respectively. The authors used this assumption for the data cache as well, thus, the latency of the fetch stage of a memory instruction would depend on both instruction and data caches.

In terms of the code under analysis, the authors assumed that it is supposed to run in isolation and contains no system calls. These assumptions are very common

in the WCET research community.

The scheme of the EVT application proposed by those researchers is based on empirical set-up of the following parameters:

- $N_{current}$ – the initial number of runs of the analyzed code;
- N_{delta} – the size of a step for increasing the number of runs;
- P – the number of distinct paths in the control-flow graph of the code under analysis;
- *difference threshold* – the parameter for estimating the difference between the EVT distribution that corresponds to $N_{current}$ runs and the EVT distribution that was constructed for $N_{current}+N_{delta}$ runs;
- *consecutive iterations counter* – the parameter for incrementing the number of consecutive iterations that satisfy some requirement specified below.
- *iteration threshold* – the parameter for taking a decision whether the process of searching for a safe EVT distribution can be stopped.

As a pre-processing phase of that scheme, the authors proposed to perform $P \times N_{current}$ runs ($N_{current}$ runs per each path) of the analyzed code and set the value of the variable that holds $N_{current}$ to be equal to $P \times N_{current}$. Then, the high-level outline of the process of applying the EVT can be considered as a loop that executes until the initially null *consecutive iterations counter* is less than or equal to the *iteration threshold*. Every iteration of this loop includes a sequence of the following phases:

Phase 1. Running the analyzed code for $P \times N_{delta}$ times (N_{delta} additional runs per each distinct path);

Phase 2. Constructing two probability distribution tail projections: for $N_{current}$ runs and for $N_{current}+P \times N_{delta}$ runs;

Phase 3. Comparing the difference between the two EVT distributions constructed during the previous phase against the threshold (difference threshold). If the difference does not exceed the threshold – the *consecutive iterations counter* should be incremented by 1. Otherwise, the *consecutive iterations counter* should be set to 0. In any case, at the end of this phase the variable that holds $N_{current}$ should be set to $N_{current}+P \times N_{delta}$.

The intermediate Phase 2 can in its turn can be represented by the following two sequential steps:

Grouping converts the frequency distribution measured during the runs into the worst-case distribution. Such conversion, is needed because of the fact that the continuous distribution function is going to be applied to discrete values of the execution time. In their work, the researchers opted for the Block Maxima method.

Fitting derives the parameters of the Generalized Extreme Value distribution, namely: the *shape*, the *scale* and the *location*. In their work, the authors used the parameters estimation [68] of the Gumbel distribution.

For finding the difference between the EVT distribution functions in Phase 3, the researchers used the *continuous rank probability score* – the probabilistic scoring rule that evaluates cumulative distribution functions operating on the same value domain [33].

When the last iteration of Phases 1-3 is performed and the corresponding loop is left behind, the authors proposed to perform the inverse cumulative distribution function's *tail extension* – the computation of the pWCET estimates that correspond to the given exceedance probabilities subject to the GEV distribution function with the final values of the parameters derived during the process discussed above.

The researchers highlighted that the resulting pWCET estimates are valid only for those P paths that were considered in the analysis. This fact links the presented

scheme to the classical *path coverage problem*. In other words, for complex codes, characterized by a huge number of possible paths, the tractability issues of applying this scheme might arise.

So far, we discussed timing analysis and parallelism in the context of uniprocessors or identical multiprocessors. However, often, systems with general-purpose processors also employ co-processors, on which it is possible to run specific tasks or portions of code much faster.

3.7 Timing analysis of architectures with co-processors

In such heterogeneous systems, certain work is still done on the main processor(s) while other work is delegated to the specialized co-processor that is dedicated to that particular type of computations. This setup is of particular interest to us because it often corresponds to how graphics processors are used.

A software task executing on a CPU that launches a remote operation on a co-processor can either (i) *busy-wait* for the duration of the operation or (ii) *self-suspend* and only resume its execution on the CPU after the results of remote operation become available. Many designers opt for the second arrangement, because it is more efficient, in that it allows the processor to be used for other ready tasks, in parallel with the co-processor operation. Unfortunately, this has the side effect of violating on of the key assumptions of the "Liu and Layland" computational model, which explicitly assumes that tasks may not voluntarily self-suspend [126]. This has necessitates new worst-case response time analysis techniques for systems with self-suspending tasks, some of which we briefly discuss below.

As mentioned earlier, in GPU computing, the GPUs are often used as co-processors. Thus, our work ties in to the real-time research that depends on the latencies of co-processor operations as input – which is what the work described in this thesis computes.

3.7.1 Suspension-oblivious approach

A simple approach which manages to remain compatible with the "Liu and Layland" model is to disregard self-suspensions and treat them as execution on the processor. For example, a task that executes for X' time units on the processor, then self-suspends for G time units, and subsequently executes for another X'' time units on the processor is modelled as a task that executes for $X' + G + X''$ time units entirely on the processor. This is simple, but potentially too pessimistic. Note also that it still requires upper bounds on the length of the self-suspending regions to be known.

3.7.2 Suspension-aware approaches

To improve on the suspension-oblivious approach, over the years, many researchers have attempted to account for the self-suspensions in the analysis. However, this has proven quite tricky, as it has recently been realized that much of the state-of-the-art is plagued by errors. This has to be borne in mind throughout the rest of this discussion and we will next point to both the original works and the corresponding fixes, where applicable. Additionally, we are aware of the fact that many researchers working on the timing analysis of self-suspending tasks are currently working on a survey, soon to be submitted for peer-review, which among other things aims to summarize problems in the state-of-the-art in the area [40].

In dissertation [110] the limited parallel model was introduced. This model considers a single general-purpose processor that delegates workloads to multiple application-specific co-processors (possibly implemented in reconfigurable hardware). The effects of bus contention are ignored and it is additionally assumed that a co-processor is not shared by different tasks. The entities of computation in the limited parallel model are software/hardware processes. The "mixed" nature of the software/hardware process is reflected by the ability of being scheduled (according to fixed-priority scheme) on the general-purpose processor, but also to issue the hard-

ware operations to the specialized co-processor. During the time that a hardware instruction is performed, the general-purpose processor may be used to execute another pending process. Hence, some process under this model may be executed by general-purpose processor in parallel with multiple processes that are executed by available co-processors.

In [5] the worst-case response time analysis of Liu and Layland [126] was generalized to be applicable to such systems. Unlike the suspension-oblivious approach, the execution of tasks on a co-processor is subtracted from the overall execution time of a task for a tighter estimation of the worst-case interference. However, this means that the worst-case scenario is no longer a critical instant (e.g., all tasks arriving at the same time). The corresponding worst-case scenario is identified in [5] and it is analogous to modelling each interfering task as having a release jitter. This accounts for the potential variability in the location of processor execution and co-processor operations inside a job activation. However, much later, it was identified, via counter-examples, that the jitter terms uses were in fact unsafe, so a fix was published [31].

In [30] the same authors published tighter analysis for linear tasks that consist of a known fixed interleaved sequence of local processor executions and self-suspending regions. The same flaw was present, inherited from [5]. It is also fixed in [31].

Jane Liu [135] analyzed tasks with self-suspensions by treating the remote operations as blocking. Recently, Chen et al. [39] provided definitive, rigorous proof for that result.

Cong Liu et al. studied self-suspensions in the context of multiprocessors with global scheduling policies, mainly for soft [130, 131, 133, 134, 129] but also for hard [127] real-time systems. To account for the temporal variability in the initiation of self-suspensions, the authors mostly rely in the concept of carry-in interference in this line of work. *Carry-in interference* is defined as interference by jobs released earlier than the job under analysis, but whose absolute deadlines are earlier than that

of the latter. Recently an errata [128] was filed for [127] by its authors.

Lakshmanan et al. also worked on the scheduling and schedulability analysis of tasks with self-suspensions [111, 113], treating the latter as blocking; some open issues with the safety of those results are discussed in [40].

Among more recent works in the area, Kim et al. [108] target the same task model as [30]. Nelissen et al. [148] identify the exact worst-case scenario for a uniprocessor system with a single self-suspending task, executing at the lowest priority. Huang et al. study fixed-priority systems [93, 94] with both linear tasks (as in [30]) and floating self-suspending regions (as in [5]).

Note that this discussion of works on self-suspending tasks is by no means exhaustive. That would have been beyond the scope of this thesis. For a fuller overview, we refer the reader to [40]. Still it becomes clear, from the works mentioned, that there is a multitude of techniques that require as input what the approaches developed under this PhD output, in the context of systems which use GPUs as co-processors.

3.8 GPU performance analysis for the average case

The GPGPU developer community has done some work on optimizing general-purpose GPU-code to achieve higher throughput [76], but usually not from a theoretical approach, but rather from an empirical/engineering perspective. On the other hand, academic work on GPU performance modelling involves rich analytical models. Ryoo et al. [168] developed two metrics (assuming non-memory intensive applications) to be used to find better configuration of a GPU source code based on the assembly-like PTX commands and resource usage information extracted by the `nvcc` compiler without complete recompilation (by the CUDA runtime) of the source code. Works [90] and [9] build models of GPU architectures to predict average-case execution time and then run the benchmarks to support their adequacy. Hong et al. [91] estimated the cost of memory requests by finding the maximum number of threads, waiting for the

data from memory, that can execute together in parallel.

A stochastic model [10] of the GPU memory system was proposed by Baghsorkhi *et al.* to monitor the average-case performance of the device, relying on Monte Carlo methodology for non-predictable aspects of the problem. Schaa et al. [170] estimated the execution time for a GPU application with multiple identical GPUs, assuming that the corresponding execution time in the case of a single GPU could be obtained empirically. Zhang *et al.* [186] target finding the bottlenecks in the performance by running benchmarks first, and only then deriving paramentrizable models to account for the latencies of the instruction pipeline, on-chip and off-chip memories. For inspecting the number of instructions and their type the authors do not rely on PTX, but on GPU simulator Barra [44] which was configured for NVIDIA GeForce 200-series GPUs. The technique highlights the sections of the low-performance code, so that the designer can tweak them afterwards.

However, all of the works mentioned above consider the execution time in the average case while, for real-time systems, we need to focus on the worst-case behaviour.

3.9 GPUs in real-time research

Heavy data-parallel workload is becoming common in modern embedded systems, hence, there is a need of massively parallel processing to make the job done. This is why the real-time systems research community demonstrates strong interest in both theoretical and practical aspects of the usage of manycore processors. Often manycores are considered as co-processors to which traditional CPUs delegate workload of a specific kind. We believe (as does Lisper [125]) that high-performance data-parallel tasks in future embedded systems will be delegated to specialized co-processors, to be run in parallel on many of their cores. This would require new timing analysis techniques, resource management frameworks and data-transfer techniques tailored for the emerging architectures. The challenge of such development is determined by

substantial architectural differences with regards to traditional Central Processing Units (CPUs).

3.9.1 GPU resource management

Bautin *et al.* developed GERM [20] – a fair-share GPU scheduler integrated into the device driver. Kato *et al.* created TimeGraph [104], RGEM [103] and Gdev [105]. TimeGraph is a fixed-priority scheduler dedicated to rendering workload, which enhances isolation and resource sharing schemes. RGEM addresses the non-preemptiveness of transfers between CPU main memory and GPU main memory, with the focus on the problem of the blocking of a higher-priority GPGPU task by the memory transfer of a lower-priority GPGPU task. The idea of the approach is to develop a user-space GPGPU runtime subsystem that splits memory transfers into multiple smaller blocks and provides preemption opportunities between these blocks. Therefore, it specifies the upper bound on the blocking time as a duration of the memory transfer of a whole single block.

Moreover, RGEM launches GPU kernels of different GPGPU tasks according to their priorities. However, once the kernel is launched it cannot be preempted which can lead to the following blocking scenario: the kernel of a lower-priority task launched earlier, can postpone the execution of the kernel of a higher-priority task. The work presents the derivation of upper bounds on such blocking delays for the sake of using them as input into traditional fixed-priority response time analysis [6]. However, this derivation assumes that the WCETs of the kernels under consideration are given as parameters.

Gdev addresses GPU resource management implemented in the OS space. Similarly to TimeGraph, it uses interrupts to schedule GPU contexts to use GPU resources. Gdev provides an API for sharing GPU main memory between GPU contexts and enables GPU resource isolation by mapping the single physical GPU to

multiple virtual GPUs to be available for OS users.

Membarth et al. proposed [145] a scheduling framework for the dynamic-priority and fixed-priority scheduling domains. The framework is supposed to be provided with the estimates on the WCET of the tasks. These estimates are obtained by taking the maximum over 100 measurements for every task. The measurements are made by using cudaEvent* functions which are relatively invasive in terms of execution time impact.

Elliott et al. [55] consider GPUs as shared resources. Their GPU management framework contains, among others, an execution cost predictor that is responsible for estimating the execution time of the real-time jobs, which is based on the past behaviour of the jobs.

Mangharam et al. [142] discussed the runtime scheduling on anytime algorithms for real-time systems. The estimation of the GPU kernel execution time is still derived from the empirical results but their schedulers are designed to adapt to the variations in actual execution time, as they are observed at run-time.

Most of the works mentioned above assume an existing data communication scheme which is characterized by the fact that the state-of-the-art GPU computing ecosystem is independent from the input/output device drivers. In particular, the data transferred between a GPU and an input/output device have to travel via CPU main memory.

3.9.2 GPU data transfer

The high-level principles of this traditional GPU data transfer scheme are the following:

i: the data is accumulated in the buffer of the device and transferred to the buffer allocated in the OS kernel space of the CPU main memory;

ii: the data has to be transferred from the “OS kernel buffer” to the buffer allocated

in the OS user space of the CPU main memory;

iii: the GPU computing application is able to access the data placed in the "OS user buffer" and copy it to the GPU main memory.

Fujii et al. pointed out [66] the importance of efficient GPU computing data transfer by specifying multiple drawbacks of this data communication scheme. Since the same data is copied multiple times between different memory areas, the whole computing system experiences additional transfer latency. By copying the same data in multiple intermediate buffers, the amount of available CPU main memory is decreased. Also, the CPU has to wait until the buffer will be filled and then has to spend cycles for copying data from one buffer to another. These reasons motivated the community to create more efficient data transfer schemes [102], [149].

Kato et al. proposed [102] removing stage (ii) by accessing the data directly in OS kernel space. This is done by mapping buffers allocated in GPU main memory to "OS user buffers", hence, when the data is copied from "OS kernel buffer" it goes directly to the GPU main memory. Thus, this method allow to reduce the number of data copies.

Nguyen et al. targeted [149] more specific problem of one-way data transfer from the Network Interface Controller (NIC) to the GPU. For the sake of simplification, the researchers assumed that the packets transferred from the NIC do not need to pass from the TCP/IP protocol stack to get an original form that it had before being sent. This simplification allowed to remove both stage (i) and stage (ii). It is achieved in a following manner: GPU computing application allocates a buffer in GPU main memory and obtains its physical address, then the address of this buffer is passed to the NIC driver and its Direct Memory Access (DMA) controller can use it for the direct data transfer from the NIC to the GPU. Hence, for this specific scenario of NIC-to-GPU transfer, the number of data copies is reduced from three to one.

An important motivation for creating more efficient data transfer schemes is that

it helps to amortize the data traveling costs in the case of the applications where the performace is crucial. For example, this allows to make the GPUs suitable for massively parallel signal processing workloads discussed below.

3.9.3 GPUs in cyber-physical systems

Cyber-physical systems often have to perform computationally expensive algorithms to monitor and control complex physical world phenomena at high speed. An example of such a system is an autonomous vehicle. To drive such a vehicle, the respective computing system should receive sensor data, process it and change the direction and speed to bring the vehicle to the correct destination and avoid accidents.

Typically, the number of sensor data that has to be processed by the autonomous driving systems is very large and is amenable to data-parallel processing. Hence, such systems could potentially benefit from GPU computing. Both the academia and the industry make much effort to materialize this idea. For example, Glavtchev et al. work on a speed-limit sign recognition system that would be part of driver support solutions for high-end automobiles [69]. This service seems to perform complex (including massively parallel) computations using CPU and GPU in the background and only notifies the human user in special important situations. Gouiffès et al. dwell on the more general problem of real-time robust obstacle detection [75].

Hirabayashi et al. addressed [88] a problem of vehicle detection targeting autonomous driving systems based on passive camera sensors. To tackle this vision-based problem the researchers opted for histograms of oriented gradients with deformable models [60]. This highly recognized approach for object detection is known to be computationally expensive which poses a serious implementation challenge, since in autonomous driving the processing should be performed in real time (in the work under consideration, it is about maintaining a frame-rate of 10-20 frames per second with 10 million of computational code blocks per frame). Assuming that the

machine learning phase required for constructing the models of a vehicle is done a priori, the authors analyzed CPU implementations [150] to identify the parts of the code that downgrade the computation performance. The workload of the respective computationally intensive blocks of code was then delegated to the GPU. Through the experiments with automotive software, the authors show the speedup obtained by their solution when executing on a computing system powered with a commodity GPU. They also quantify the performance characteristics that should be achieved to allow the approach to become a candidate for integrating in real-life autonomous driving systems.

The works discussed above either assume that GPU execution requirements are given, or obtain the respective estimates in some straightforward way. However, an important aspect of the usage of the GPUs in varios (including cyber-physical) applications, is that it requires bounds on the execution requirements of the computation entities. Depending on the strictness of timeliness guarantees required, these bounds might result from applying the techniques of different safety levels. In any case, GPU timing analysis is of paramount importance for successful integration of GPUs in the applications that have some temporal constraints.

3.9.4 GPU timing analysis

In GPUs, entities of execution (threads) share computational and load/store units within a processor, therefore, threads greatly affect each other while executing in parallel. Unlike for CPUs, which are latency-oriented processors, the worst-case execution time of a single thread is not that important for GPU timing analysis, therefore, analyzing the latency of a particular thread execution is not a primary goal. Being designed for rendering purposes, GPUs are throughput-oriented processors, since it is the common execution of many threads that gives the result. The aspects mentioned above distinguish between these two processor architectures, and cause substantial

difficulties in applying already well-established CPU timing analysis techniques to GPUs. Hence, related work on GPU timing analysis is rich with unrealistic assumptions and simplifications which reflect the hardship the researchers (us included) are inevitably facing in their work with GPUs.

Betts *et al.* presented [28] two WCET techniques for estimating CUDA kernel functions running on GPU simulator GPGPU-sim [11]. However, the fact that NVIDIA GPUs are switching from hardware to software implementation for their scheduling stage (responsible for register scoreboarding and dependencies checking) makes it even less feasible to rely on the third-party GPU simulators.

The first technique (called "dynamic") uses measurements to estimate the worst-case release jitter of the latest warp and to estimate the warp-specific WCET, assuming that the latter number should include timing effects of multiple streaming multiprocessors competing for shared resources (e.g., L2 cache, GPU global memory bandwidth, etc.).

The second technique (called "hybrid") assumes a constant time delay for the release of every warp and uses static analysis based on instrumentation point graphs which is supported by the parameters obtained from measurements. A pivotal assumption of the static part of the technique is that the warps arrive in waves, where a subsequent wave of warps cannot be processed until the latest warp of the previous wave is completed. The warp constant time delay, the number of waves and the size of a wave are supposed to be obtained by measurements. Implicitly, the authors assume that the warps in waves are scheduled according to a round-robin scheduling policy, which is probably a simulator-based assumption. The authors took into account only the GPU kernel execution time, not considering the timing analysis of the CPU code that allocates data structures in GPU main memory and copies data from CPU main memory to GPU main memory and back.

Still, it is important to note that, static instrumentation point graphs tend to be

pessimistic; conversely, high-water mark times may be optimistic, if no methodology for deriving safe upper-bounds is applied.

Hirvisalo presented [89] a theoretical GPU model and a static timing analysis approach inspired by the Cooperative Thread Arrays [7] (CTA) assuming that it implements only a single thread block as a set of warps of parallel threads. The approach includes the following phases: a static control flow divergence analysis, an abstract warp construction, an abstract CTA simulation. The author assumes that the static divergence analysis will represent every warp as a sequence of instructions to execute. With the help of assumptions that the warps are scheduled according to the round-robin scheduling, the next phase of the approach is responsible for the construction of an abstract warp – an oriented graph which is aimed to abstract away from multiple warps and represent all of them with a single entity. Due to the scheduling assumption, there is no need to consider how long it takes any given warp to execute a particular instruction by every warp, which warp is the earliest one to execute that instruction and which warp is the latest one. Instead, any basic block of an abstract warp is characterized by an upper bound on the execution time of the instruction performed by all warps that take the corresponding control-flow path. This holds as in round-robin scheduling, an instruction is executed in convoy – the sequence of eligible warps all performing that instruction in a round, however, for another kind of scheduling approach (e.g., Most Pending Warp Executes First [23]) the concept of abstract warp would not be applicable. The GPU model assumes only a single streaming multiprocessor available on chip with a low access latency to the main GPU memory, an absence of an on-chip memory subsystem (e.g., caches) and an availability of the kernel code in a simplified assembly form with well-defined timing characteristics. The phase of an abstract CTA simulation is conducted by an algorithm for traversing abstract warps, subject to an assumptions that the loop-bound analysis of the kernel code would always provide an exact value of the loop

iteration number and an absence of parallelism of basic block execution by multiple warps. Implicit simplifications include assumptions that there is no contention between warps for the computational and load/store units, single memory transaction for the case of memory transfers from/to GPU main memory and absence of dynamic parallelism [154].

This brings us to the research conducted within the framework of this thesis. This research includes the approaches from both branches of timing analysis: measurement-based (see in Chapter 7) and static (see in Chapter 5 and Chapter 6). The latter approaches are based on the GPU programming model and the model of GPU architecture discussed next.

4 GPU model

The development of static timing analysis approaches requires the model of GPU architecture and the GPU programming model that are amenable to analyzing. Our models are based on the GPU technology from NVIDIA. Thus, we are going to use the terminology introduced by this chip-maker. However, given that the technologies from other GPU vendors have strong similarities with the one from NVIDIA, the considerations presented in this thesis can be applied to those GPUs as well.

In terms of outline, Section 4.1 discusses the GPU programming model, Section 4.2 introduces the model of GPU architecture, Section 4.3 summarizes the most imortant considerations and assumptions that should be kept in mind while reading Chapter 5 and Chapter 6 of this thesis.

4.1 GPU programming model

Novel parallel programming models, such as Nvidia CUDA [156] and OpenCL [107], brought us to the GPU computing: the general-purpose use of GPUs for the broader range of workloads, not just graphics. These programming models utilize the strenghts of those design concepts implemented in the GPUs. GPUs are designed for high throughput via massive parallelism; not via executing any single thread particularly fast. Therefore, the applications best-suited for GPUs: (i) are easily decomposable in thousands of parallel threads; (ii) have minimal dependency across data (no need for synchronisation; maximum parallelism); (iii) are computationally intensive, to justify the costly copying of the GPU input and output over the bus. In many cases, the GPU is used as a *co-processor* to which certain functions are offloaded for speed up – and this is the use we are most interested in this thesis.

The theoretical basis of the corresponding programming models was established by *stream processing*. The stream processing computational paradigm was conceived

so as to allow efficient processing for a particular type of parallel applications (with minimal data dependencies) while simultaneously simplifying the parallel hardware architecture. Given a set of data (a *stream*), a series of operations (*kernel function*) is applied to each element in the stream. This paradigm applies very nicely to graphics and was partially implemented in GPUs. In other words, GPUs were designed to execute a large number of threads (in the order of thousands or more) so that their joint execution provides a result to a user. Therefore, in terms of timing analysis, we are not interested in one particular thread but in a group of many threads whose joint execution provides the result. Hence, the focus on the *worst-case makespan* – the longest possible time interval from the moment when the "earliest" thread starts executing, until the "latest" thread terminates.

However, it is important to emphasize that GPU threads differ greatly from CPU threads as the respective hardware architectures are drastically different. CPUs have branch prediction (so that a thread does not have to wait for the result of a branch), speculative execution (so as to perform computations before even being sure if the result will be needed), out-of-order execution (wherein an instruction can be performed as soon as its operands become available), substantial cache hierarchy (so as to read/write the data faster in the average case), prefetching (to get the data earlier).

All these hardware optimizations, that CPUs are built around, aim to minimize the average latency. In contrast let us consider a GPU-thread which is running and needs to access the main memory. It takes hundreds of clock cycles to do that [158] and the GPUs do not have such a sophisticated architecture, like the one earlier described, that would help run a thread faster. Therefore, whenever the GPU thread sends a request to the main memory, the processor switches to executing another thread. In the general case, whenever any GPU thread stops for some reason, if there is enough work to do, we can always keep the streaming multiprocessor busy in the meantime. In this way, throughput is good, even if the processing of a single thread

is not always fast. Instead of minimizing latency (like CPUs do), GPUs have a large number of computational units and switching between threads "hides" the latency and consequently increases the efficiency.

Another important aspect is that GPU threads are much more "light-weight" than ordinary CPU threads, because context-switching between them does not involve updates to operating system data structures and takes very few clock cycles. One of the reasons that context-switching between GPU-threads is fast is that all of them execute the same program (the "kernel"[5]) in parallel. This is also why, in GPU computing, it is much more convenient to think not in terms of individual threads but, instead, in terms of another entity of computation, the *warp*[6] – a group of threads, each of which executes the same kernel concurrently.

At run-time, warps are bundled together in groups termed *thread blocks* and each thread block is sent to one streaming multiprocessor for execution. Each streaming multiprocessor has a few thread blocks assigned to it at any time. Thread blocks do not migrate among streaming multiprocessors. The CUDA engine tries to keep the processing units of each streaming multiprocessor busy but exactly how warps are dispatched is not publicly documented.

The concept of a *thread block* (*work group* in the terminology of OpenCL) has similarities with the independent thread model (see Section 3.3.2). Traditionally, the chip-makers do not provide an API for synchronizing thread blocks, thus by default their execution is synchronized only by the start of the kernel and by the termination of the kernel. However, there exists an approach presented by Feng et al [61] to make an implicit synchronization based on atomic operations. Although it is sophisticated from the engineering viewpoint, this approach is not considered to be a good development practice since it requires an awareness of the chip organization details that are usually kept as a secret by the chip-maker. Hence, such an implicit synchronization

[5]not to be confused with operating system kernels
[6]AMD ATI GPUs have a similar concept to that of a warp called a wavefront [1]

is considered to be non-generic and to require considerable implementation effort including reverse engineering. Moreover, the use of atomic operations not only limits the performance of the application under consideration, but also increases the risk of making an error in the code which would be hard to debug. On the other hand, the unit of scheduling is not the thread block, but the *warp* (*wavefront* in OpenCL) – a smaller group of threads that execute all together in parallel.

The concept of the warp introduces [124] a Single Instruction Multiple Threads (SIMT) parallel execution model which has similarities with the concept of gang scheduling (see Section 3.3.1). Every GPU thread has dedicated registers (including the program counter). Thus, from the high abstraction viewpoint, it allows GPU threads to follow different paths. Still, a thread is scheduled only as an element of the warp, where all threads execute in a lock-step. Therefore, if there is a divergence in their control flows, threads that are following the same path will execute while the others will bet idle and vice versa. In such a way, the parallelism is limited until the point of the convergence. It is important to note, that GPU execution makes sense when only a few threads diverge in control flow. In the case that many threads diverge – the data-transfer overhead would be hard to amortize.

4.2 GPU architecture model

Our analysis considers a streaming multiprocessor inspired by NVIDIA Kepler [154] and NVIDIA Fermi [152] – hardware architectures of GPUs. Each streaming multiprocessor has a relatively complex structure, which makes its timing analysis a non-trivial open problem. Therefore, in this work, we restrict our focus to the timing analysis of a single such streaming multiprocessor.

4.2.1 Streaming multiprocessor

The streaming multiprocessor(see in Figure 5) includes (i) multiple CUDA cores that are capable of boolean, integer and floating-point arithmetic, (ii) multiple "load/store" units that load data from and store data to cache or DRAM, (iii) multiple special function units that implement computation of sine, cosine, square root and boolean inverting directly in hardware and multiple double-precision units that are responsible for 64-bit arithmetic. GPUs evolve fast (even by chip makers industry standards), hence, the configuration of the streaming multiprocessors of different GPU models often varies (although these GPUs could belong to the same generic GPU architecture). The configuration of a GPU-chip (particularly the number of computational units of each kind in a streaming multiprocessor) is specified by the term *compute capability* [158] – an identifier in the format $x.y$ where $x, y \in \mathbb{N}$.

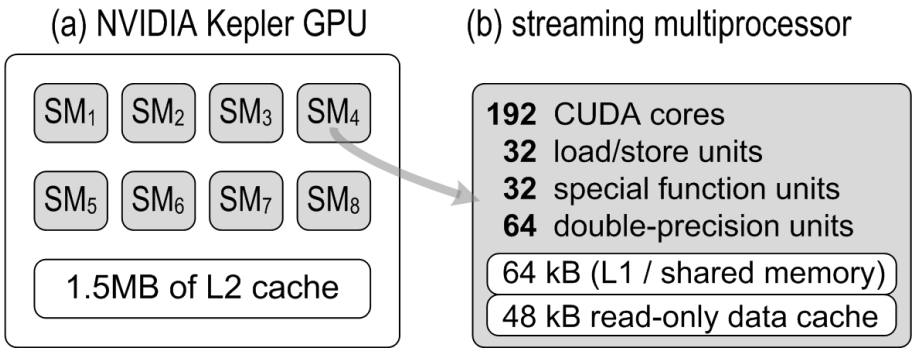


Figure 5: A simplified scheme of the NVIDIA Kepler GK104 GPU chip that contains 8 streaming multiprocessors.

In GPU multiple lightweight threads are advancing together in parallel subject to the capacity of shared computational resources of a streaming multiprocessor. For example for the GPU device of compute capability 3.0 the streaming multiprocessor includes 192 CUDA cores and 32 load/store units. Therefore, it is possible that 192 threads perform arithmetic operation concurrently but only 32 of them can store data

in parallel with each other.

4.2.2 Entities of computation

A streaming multiprocessor processes warps, while its CUDA cores and load/store units process the corresponding threads. All threads of all warps, which are running on a given streaming multiprocessor, execute the same kernel [152].

The maximum efficiency occurs when the threads of the warp, all together in parallel, follow the same execution path. However, every individual thread has its own execution context (instruction address counter, states of registers, etc) therefore it is able to execute and branch independently of other threads within the same warp. Since warps execute independently, regardless of whether they are taking the same path or not, talking about control flow divergence makes sense only for threads within a single warp. If the threads of the same warp branch in different directions, the hardware sequencer keeps track of the diverged threads. It broadcasts the instruction fetch to the computational units that serve the threads of the same branch. Upon reaching the point of convergence, the threads stall waiting for the threads of the other branch, so that they can resume the execution of a common instruction together in parallel.

A streaming multiprocessor manages, schedules, and executes warps. The scheduling engine of a streaming multiprocessor comprises several warp-schedulers each of which includes few instruction dispatch units. Given warps to execute, a streaming multiprocessor allocates them among its warp-schedulers. Then at instruction issue time each warp-scheduler selects an active warp (one that has threads ready to execute its next instruction) and issues few independent instructions from corresponding threads. The number of the instructions that could be issued for the particular warp is bounded by the number of instruction dispatch units of the warp-scheduler and by the number of corresponding computational units (to process these instructions)

available. Therefore, if the warp-scheduler includes δ instruction dispatch units, up to δ instructions (that have no dependencies between each other) could be performed concurrently.

4.2.3 Simplifying assumptions

The execution context of any thread is stored in the on-chip memory [158] as long as the corresponding warp exists, therefore, switching from one context to another is lightweight. The term *instruction latency* specifies the number of clock cycles it takes for a warp to execute a given instruction. Full utilization of the streaming multiprocessor is achieved when there is enough workload to keep all its computational units continuously busy. For example, when a warp is stalled on I/O, the streaming multiprocessor quickly switches to another warp (in a single cycle). This technique is known as "latency hiding".

According to our simplified model, all the data has to be stored in a single-type memory and we assume that the data layout does not influence the latency of the corresponding memory instruction. Moreover, for the sake of simplification, we assume that all the data needed is already present in level-1 cache, thus, the data access latency is minimal. For the sake of clear presentation all instructions under consideration are supposed to require only a single clock cycle for their execution. However, later in Section 4.2.5, we present a technique for modelling instructions that have multi-cycled latency.

As stated earlier, the warps are competing for the computational resources of a streaming multiprocessor according to some largerly undocumented scheduling policy. The chip-maker reported [153] about the move from complex scheduling logic implemented in hardware (as it is done in NVIDIA Fermi) towards software scheduling that is performed at run-time (in NVIDIA Kepler). However, we still do not have concrete publicly available information about the actual scheduling policy. Similar to

[22], in this work we therefore simply assume that the scheduling is work-conserving: whenever there are warps available and free computational units in a streaming multiprocessor, these units are used to execute some warps.

A streaming multiprocessor under our model includes σ warp-schedulers and each of them comprises only a single instruction dispatch unit, therefore, we assume that any warp-scheduler is able to issue no more than one instruction per clock cycle. Hence, the number of warps that could be processed in parallel by a single streaming multiprocessor is pessimistically bounded by σ and the overall computational capacity of a streaming multiprocessor. According to the information available [154] there is a pair of instruction dispatch units per each of 4 warp-schedulers of a streaming multiprocessor in NVIDIA Kepler. However, in our model we pessimistically restrict the number of instruction dispatch units per warp-scheduler to avoid the difficulties of having dependencies among the instructions that were dispatched by multiple units in parallel. We also pessimistically assume that all the types of computational units in a streaming multiprocessor under consideration are not pipelined.

We assume that there is no off-chip data traffic. This is an optimistic assumption (which ought to be relaxed in future work) but partially justified by the fact that in GPU architectures under consideration the amount of the on-chip memory is relatively big [152], [154]).

4.2.4 Kernel instruction string

Early works [73], on using GPUs for general-purpose computation, contain a lot of reverse engineering efforts and in terms of programming, everything was developed by hand in assembly code. The positive aspect of the low-level coding was that the developers had better knowledge of how their programs would use the hardware units of the GPUs. Later, researchers began to use the OpenGL graphics interface [162] for general-purpose computation. This was also tedious because, although in most

cases the code did something completely different, it still had to be written as if it were graphics computations.

Nowadays, the programming model for GPU computing is moving towards that of the high-level programming languages. CUDA not only provides users with the APIs for high-level programming languages (C, C++, Fortran, wrappers for Java and Python), support for computational interfaces (OpenCL, DirectCompute) and for directive-based OpenACC, but it also specifies a virtual Instruction Set Architecture (ISA) which is kept relatively stable over the generations of the GPUs developed by NVIDIA. This ISA, the pseudo-assembly language and the low-level virtual machine are all called PTX because they were designed for *parallel thread execution*. Since GPUs evolve rapidly, via PTX, NVIDIA provides a stable layer of pseudo-assembly language to developers, while remaining free to change the underlying instruction set later, if necessary.

The high-level GPU-code is processed by a specialized compiler (that supports the extensions that CUDA adds to programming languages); the one from NVIDIA is called `nvcc` [158]. Running this compiler with the `-ptx` flag will output the human-readable representation of the pseudo-assembly code that is put into an object file. This file serves as input to the CUDA-driver which includes another compiler that translates the PTX-code into the target ISA – a binary code that can be run on a particular hardware. Although PTX-code is not the machine code that is actually executed by the hardware, we (like Ryoo et al. [168]) rely on it for the purposes of counting the number of the instructions and their mix. Given that we are interested in the usage of the computational units of a streaming multiprocessor, we abstract away from the assembly code using the kernel instruction string [22] – a sequence of "*L*", "*C*", "*S*", and "*D*" symbols, each of which represents a hardware instruction that should be performed on load/store unit ("*L*"-instruction), CUDA-core ("*C*"-instruction), special function unit ("*S*"-instruction) and double-precision 64-bit unit

("D"-instruction). For example, the kernel instruction string "LS" specifies that an instruction should be carried out by the load/store unit, followed by an instruction that should be performed on the CUDA core.

Clock Cycle	1	2	3	4	5	6	7	8
Warp 1	L	C			L			
Warp 2		L	C			L		
Warp 3			L	C			L	
Warp 4				L	C			L

Figure 6: Possible schedule (round-robin, $\sigma_L = \sigma_C = 1$) as a valid solution

4.2.5 Architectural details

Since our goal is to make the model as generic as possible addressing GPUs of different compute capabilities, we introduce a set of variables to specify the configuration of a streaming multiprocessor. Assume that the streaming multiprocessor includes computational units of some generic type U, let us define the variable σ_U as it is in equation (1)

$$\sigma_U = \frac{uUnitsNumber}{warpSize} \tag{1}$$

Given that $uUnitsNumber$ equals to the number of u-units that a streaming multiprocessor includes, and $warpSize$ equals to the number of threads per warp, σ_U specifies the maximum number of warps that can perform "U"-instruction within the same clock cycle on a single streaming multiprocessor. However, if the number of computational units of some kind is less than the warp size, it is not possible to execute corresponding instruction by all threads of the warp within a single clock cycle. To illustrate, for the possible schedule (in Figure 14) this means that at every

single column (that corresponds to a clock cycle) the number of "U"-symbols cannot exceed σ_U. As an example let us consider a streaming multiprocessor of compute capability 2.0 that has 32 threads per warp, 32 CUDA-cores, but only 16 load/store units, therefore

$$\begin{aligned} \sigma_L &= \frac{lUnitsNumber}{warpSize} = \frac{16}{32} = \frac{1}{2} \\ \sigma_C &= \frac{cUnitsNumber}{warpSize} = \frac{32}{32} = 1 \end{aligned} \tag{2}$$

In such case, 16 threads of the warp (a *half-warp* [158]) will execute an "L"-instruction in one clock cycle, and another half-warp will execute this instruction in a later clock cycle. At the cost of some pessimism this could be considered to be equivalent to "L"-instruction latency of two clock cycles. To simplify the analysis by having the same instruction latency for every type of instruction we transform the kernel instruction string (e.g., "LC", $\sigma_L = \frac{1}{2}$) replacing every original "L"-instruction by two consecutive "L"-instructions (getting "LLC", $\sigma_L = 1$ as a result). We can describe this transformation technique for the generic "U"-units as follows. Given that in NVIDIA general-purpose GPU-architectures (Kepler, Fermi, GT200, G80) the value of $uUnitsNumber$ is a power of 2, and $uUnitsNumber < warpSize$ implies $warpSize$ **mod** $uUnitsNumber = 0$, the value of σ_U will be fractional: $\sigma_U = \frac{uUnitsNumber}{warpSize} = \frac{1}{n}$, where $n \in \mathbb{N}$. Multiplying both sides of the equation by n, we get

$$n \cdot \sigma_U = 1$$

We can transform the instruction string for our kernel by replacing each "U"-instruction with n "U"s (each one corresponding to each "subwarp" of $\frac{warpSize}{n}$ threads) and additionally assuming that $\sigma_U = 1$ (the transformation is equivalent because n subwarps of a warp, execute the "U"-instruction in mutual exclusion [12]). For our example of a streaming multiprocessor of compute capability 2.0, where $L = 16$, $C = 32$, $S = 32$,

$\sigma_L = \frac{1}{2}$, $\sigma_C = 1$, the instruction string "LC" will be transformed to "LLC". (In other words, the original "L"-instruction is replaced by 2 consecutive "L"-instructions) and the value of σ_L will be changed to $\sigma_L = 1$ (Figure 7). We apply the same technique (at the cost of some pessimism) to the occasional CUDA instruction that takes more than one cycle.

$$\underset{\sigma_L = \frac{1}{2}}{LC} \Rightarrow \underset{\sigma_L = 1}{LLC}$$

Figure 7: Transformation of the kernel instruction string

4.3 Summary

The assumptions and the most important considerations of the section are summarized as follows:

- A streaming multiprocessor includes four types of computational units: load/store, special function, double-precision, CUDA cores, and their respective quantities are $lUnitsNumber$, $sUnitsNumber$, $dUnitsNumber$, $cUnitsNumber$.
- For the purpose of the parallelism the threads are organized into groups called warps. Each warp comprises up to $warpSize$ threads.
- All threads of all waprs of a given streaming multiprocessor execute the same kernel instruction string.
- All the data needed are in level-1 cache, therefore, we do not have to account for the latency of memory operations.
- Any instruction takes a single clock cycle and is executed in "atomic"-fashion – it holds the computational resource exclusively and cannot be interrupted.

- The warps are scheduled in a work-conserving way by σ warp-schedulers, and we pessimistically assume that only a single instruction can be scheduled from the given warp by the available warp-scheduler. Therefore, the number of warps that could be processed in parallel by a single streaming multiprocessor is bounded by the following value:

$$min\{\sigma, \sigma_L + \sigma_C + \sigma_S + \sigma_D\}$$

  A warp may be scheduled by at most one warp-scheduler at a time.

- The goal of our timing analysis is to find the worst-case makespan (further on referred to as the makespan) the longest possible time interval between the moment when the "earliest" thread starts execution, and the moment when the "latest" one finishes, subject to the given kernel instruction string and the configuration of the streaming multiprocessor.

5 Optimization-based approach

For the sake of brevity in this chapter we consider a streaming multiprocessor with only two types of computational units (CUDA cores and load/store units). However, all the considerations could be applied to GPUs with additional types of units in a straightforward way.

In the remainder of this chapter, Section 5.1 offers a new fast but pessimistic method for calculating an upper bound on the makespan for a single streaming multiprocessor. Section 5.2 considers the formulation of a binary Integer Linear Programming (ILP) problem of finding the exact value of the worst-case makespan. Section 5.3 provides an alternative optimization problem formulation. Section 5.4 summarizes the ILP derivation. Section 5.5 introduces the technique for efficiently computing a safe estimate on the worst-case makespan. Section 5.6 presents the results of the experiments. Section 5.7 concludes.

5.1 Pessimistic makespan derivation

Let us introduce an approach with very low computational complexity for deriving an upper bound on the makespan of a group of threads executing on a streaming multiprocessor. This approach is pessimistic but its output may serve as input to other, less pessimistic, derivations (as later shown).

The pessimistic derivation formulated in this subsection is based on the fact that a streaming multiprocessor is used most inefficiently when, in a given clock cycle, all warps contend for the same type of computational unit. In that scenario, the computational units of other types are "wasted" (i.e., cannot be used for "latency hiding") because they cannot be used to advance any warp in computation (during that cycle).

This can be illustrated by the following example: 128 threads (in 4 warps of 32)

all execute the same kernel (with instruction string "LLC") on a single streaming multiprocessor. Figure 8 presents one possible schedule (which is work-conserving). Note that during the first 5 clock cycles, the multiprocessor has a throughput of only one instruction per warp per cycle (Figure 8), because initially *all* warps need to perform two consecutive load/store instructions and the CUDA cores are of no use to any of them (hence remain idle).

Clock Cycle	1	2	3	4	5	6	7	8	9	
Warp 1	L				L	C				
Warp 2		L				L	C			
Warp 3			L				L	C		
Warp 4				L				L	C	

Figure 8: Possible schedule ($\sigma_L = \sigma_C = 1$)

Accordingly, our pessimistic makespan derivation assumes that for every instruction of a given warp, *all* other warps are also competing for the same computational unit, at the time of its issue. To enforce this (very pessimistic) assumption, we no longer consider the actual kernel instruction string but rather just the number of instructions of a given type in that string.

Assume that the kernel instruction string α has length I and that there are two types of computational units: load/store and CUDA (represented by "L" and "C" in the string). Then, I_L and I_C is the number of "L"s and "C"s in the kernel instruction string (i.e., $I_L + I_C = I$). From the original kernel instruction string, we derive two strings: one string α_L consisting exclusively of "L"s (I_L in count) and one string α_C consisting exclusively of "C"s (I_C in count). In equations:

$$\alpha_L = \{\underbrace{L\ L\ \ldots\ L}_{I_L\ \text{“L”s}}\} \tag{3}$$

$$\alpha_C = \{\underbrace{C\ C\ \ldots\ C}_{I_C\ \text{“C”s}}\} \tag{4}$$

The pessimistic worst-case makespan is then derived as

$$T = T_L + T_C \tag{5}$$

where T_L is the worst-case makespan for a group of W hypothetical warps executing α_L as kernel (and likewise for T_C and α_C). In turn, T_L and T_C are derived as:

$$T_L = \left\lceil \frac{W}{\sigma_L} \right\rceil I_L \tag{6}$$

$$T_C = \left\lceil \frac{W}{\sigma_C} \right\rceil I_C \tag{7}$$

5.2 ILP derivation

In this subsection we present the formulation of the worst-case makespan derivation problem as a binary ILP. The solution of the ILP instance provides the exact (subject to our simplifying assumption) worst-case makespan. In order to generate the ILP instance from the problem instance, we also employ the pessimistic makespan derivation described in the previous subsection.

Assume that the kernel (known beforehand) consists of I instructions. We can

present the sequence of the instructions using binary constants with index $i = 1..I$.

$$IL_i = \begin{cases} 1 & \text{if instruction } i \text{ is for a load/store unit;} \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

$$IC_i = \begin{cases} 1 & \text{if instruction } i \text{ is for a CUDA core;} \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

$$\forall i \quad IL_i + IC_i = 1 \tag{10}$$

It is obvious that the schedule for which the worst-case (i.e., longest) makespan is observed can be no longer than T clock cycles, where T is the makespan estimate (5) computed under the simple pessimistic approach described earlier in Section 5.1.

To describe the schedule of W warps over T clock cycles, we introduce the following binary decision variables, specifying the usage of the resources of the streaming multiprocessor:

$$LS_{w,i,t} = \begin{cases} 1 & \text{if warp } w \text{ performs instruction } i \text{ on} \\ & \text{load/store unit at clock cycle } t \text{ ;} \\ 0 & \text{otherwise.} \end{cases}$$

$$CC_{w,i,t} = \begin{cases} 1 & \text{if warp } w \text{ performs instruction } i \text{ on} \\ & \text{CUDA core at clock cycle } t \text{ ;} \\ 0 & \text{otherwise.} \end{cases}$$

where indexes $w = 1..W$ and $t = 1..T$ stand for warps and clock cycles respectively.

With the help of these variables, the formulation of the ILP is presented as follows: in Section 5.2.1 we derive the objective function, corresponding to the worst-case makespan; Section 5.2.2 formulates capacity constraints on the computational

resources of the single streaming multiprocessor; Section 5.2.3 states precedence constraints for the instructions of the kernel instruction string; Section 5.2.4 dwells on constructing constraints that guarantee the work-conserving property of the schedule.

5.2.1 Objective function

The objective function should be designed in such a way, to provide the longest possible makespan when all the constraints are satisfied. Relying on the precedence constraints between instructions, we notice that the makespan is maximized iff the last instruction of the last warp to complete (whichever that is), is executed as late as possible. Since this would be the I^{th} kernel instruction, the worst-case makespan is then given by

$$\max_{w=1..W,t=1..T}\{t \cdot (LS_{w,I,t} + CC_{w,I,t})\} \tag{11}$$

Given that the objective function of our optimization problem should be linear, we need to add some extra constraints to present (11) in a proper way. Although, in principle, we are not interested in which one of the W warps executes the last instruction in the schedule, specifying that would allow us to simplify (11). Without loss of generality, since all warps are identical, any schedule with worst-case makespan can be transformed into a schedule where the last completing warp is the warp W (e.g., via re-indexing of warps). We can express this additional requirement using $(W-1)$ constraints:

$$\forall w = 1..(W-1)$$

$$\sum_{t=1}^{T}(t \cdot (LS_{w,I,t} + CC_{w,I,t})) \leq \sum_{t=1}^{T}(t \cdot (LS_{W,I,t} + CC_{W,I,t}))$$

Therefore (11) could be presented as finding the clock cycle when the warp W

executes an instruction with the index I.

$$\max_{t=1..T}\{t \cdot (LS_{W,I,t} + CC_{W,I,t})\} \tag{12}$$

However, there exists only one $t' \in [1..T]$ such that warp W performs instruction I at cycle t'. Therefore $\forall t'' \in [1..T]$, $t'' \neq t'$: $LS_{W,I,t''}=0$ and $CC_{W,I,t''}=0 \Rightarrow LS_{W,I,t''} + CC_{W,I,t''} = 0$. Hence expression (12) can be rewritten as a linear function of $LS_{W,I,t}$ and $CC_{W,I,t}$ as follows:

$$\sum_{t=1}^{T}(t \cdot (LS_{W,I,t} + CC_{W,I,t})) \tag{13}$$

This is the objective function (that should be maximized) in our binary ILP-formulation.

5.2.2 Capacity constraints

As explained in Section 4.1, the makespan is dependent on how internal resources in a streaming multiprocessor (CUDA cores and load/store units in our case) are shared between threads. Although streaming multiprocessors of modern GPUs have many computational units, these are still finite resources. Additionally, the number of computational units of each type (i.e., L and C) is typically different. Such limitations, among others, can be represented by the following constraints:

An upper bound on the number of load/store instructions that could be performed within a single clock cycle t, could be expressed as:

$$\forall t \quad \sum_{w=1}^{W}\sum_{i=1}^{I} LS_{w,i,t} \leq \sigma_L \tag{14}$$

Similarly for the number of CUDA instructions:

$$\forall t \quad \sum_{w=1}^{W}\sum_{i=1}^{I} CC_{w,i,t} \leq \sigma_C \tag{15}$$

Any warp is able to perform no more than one instruction at a single clock cycle:

$$\forall w, t \quad \sum_{i=1}^{I} LS_{w,i,t} \leq 1, \quad \sum_{i=1}^{I} CC_{w,i,t} \leq 1 \tag{16}$$

Any instruction can only be executed on a computational unit of a specific respective type:

$$\forall w, i \quad \sum_{t=1}^{T} LS_{w,i,t} = IL_i, \quad \sum_{t=1}^{T} CC_{w,i,t} = IC_i \tag{17}$$

The constraints expressed by Equations (10) and (17) mean that:

– If $(IC_i = 1)$ then $(\forall w, t \quad LS_{w,i,t} = 0)$

– If $(IL_i = 1)$ then $(\forall w, t \quad CC_{w,i,t} = 0)$

Additionally, Equations (17) and (10) ensure that every instruction is performed by every warp.

5.2.3 Precedence constraints

Since the kernel instructions are executed in a particular order by all warps, we must model the constraints of precedence between them. For these purposes it is useful to introduce auxiliary (not decision) variable $Y_{w,i}$ which denotes the clock cycle when warp w executes instruction i. This new variable facilitates expressing the constraint that $\forall i = 1..(I-1)$ and for every warp, the instruction $i+1$ cannot be executed until after the instruction i has been executed by the same warp:

$$\forall w \quad Y_{w,1} < Y_{w,2} < \cdots < Y_{w,I-1} < Y_{w,I} \tag{18}$$

Taking into account Equations (17) and (10), one may see that $(Y_{w,i} = t)$ is equivalent to $(\sum_{t'=1}^{t}(LS_{w,i,t'} + CC_{w,i,t'}) = 1)$

That could be written as

$$Y_{w,i} = \sum_{t=1}^{T}(t \cdot (LS_{w,i,t} + CC_{w,i,t})) \tag{19}$$

By substitution of Equation (19) to (18), we get:

$$\forall w, i = 1..(I-1)$$

$$\sum_{t=1}^{T}(t \cdot (LS_{w,i,t} + CC_{w,i,t})) < \sum_{t=1}^{T}(t \cdot (LS_{w,i+1,t} + CC_{w,i+1,t}))$$

In linear programs the inequalities should be non-strict [184]. Therefore (since the decision variables are integer) we rewrite the above as:

$$\forall w, i = 1..(I-1)$$

$$1 + \sum_{t=1}^{T}(t \cdot (LS_{w,i,t} + CC_{w,i,t})) \leq \sum_{t=1}^{T}(t \cdot (LS_{w,i+1,t} + CC_{w,i+1,t}))$$

5.2.4 Work-conserving constraints

One of our assumptions, stated in Section 4.2.3, was about the scheduling policy implemented in GPU. Namely, that it is work-conserving. This means that whenever there are warps available and free computational resources on the streaming multiprocessor, the scheduler must select some warp for execution. Next, we introduce some additional variables, for the purpose of modeling the work-conserving property of the schedule via ILP constraints.

Let us assume that instruction i is for a load/store unit ($IL_i = 1, IC_i = 0$). Then $LSREADY_{w,i,t} = 1$ iff warp w was ready to execute instruction i at clock cycle t (i.e., it had already executed instructions $1..(i-1)$) but did not. Similarly with variable $CCREADY_{w,i,t}$ if $IL_i = 0$ and $IC_i = 1$. In formal notation:

$$\forall w, t$$

$$LSREADY_{w,1,t} = \begin{cases} 1 & \text{if } (IL_1 = 1) \wedge (t < Y_{w,1}) \ ; \\ 0 & \text{otherwise.} \end{cases}$$

$$CCREADY_{w,1,t} = \begin{cases} 1 & \text{if } (IC_1 = 1) \wedge (t < Y_{w,1}) \ ; \\ 0 & \text{otherwise.} \end{cases}$$

$$\forall w, i = 2..I, t$$

$$LSREADY_{w,i,t} = \begin{cases} 1 & \text{if } (Y_{w,i-1} < t) \wedge (IL_i = 1) \\ & \wedge (t < Y_{w,i}) \ ; \\ 0 & \text{otherwise.} \end{cases}$$

$$CCREADY_{w,i,t} = \begin{cases} 1 & \text{if } (Y_{w,i-1} < t) \wedge (IC_i = 1) \\ & \wedge (t < Y_{w,i}) \ ; \\ 0 & \text{otherwise.} \end{cases}$$

A schedule is not work-conserving iff there exists some warp w that is ready to perform some instruction i at clock cycle t, but stays idle, even if there were spare computational units (of the type that instruction i runs on). This scenario could be expressed as follows:

$$\begin{aligned}\exists w,t \quad & \Big((\sum_{i=1}^{I} LSREADY_{w,i,t} \neq 0) \wedge \\ & (\sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} < \sigma_L)\Big) \vee \\ & \Big((\sum_{i=1}^{I} CCREADY_{w,i,t} \neq 0) \wedge \\ & (\sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t} < \sigma_C)\Big)\end{aligned} \tag{20}$$

If and only if the expression (20) does not hold (or equivalently, its logical complement holds), the schedule is work-conserving. The logical complement to (20) can be derived via application of De Morgan's laws and is the following:

$$\begin{aligned}\forall w,t \quad & \Big((\sum_{i=1}^{I} LSREADY_{w,i,t} = 0) \vee \\ & (\sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} = \sigma_L)\Big) \wedge \\ & \Big((\sum_{i=1}^{I} CCREADY_{w,i,t} = 0) \vee \\ & (\sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t} = \sigma_C)\Big)\end{aligned} \tag{21}$$

In a system of ILP-constraints, expression (21) can be split into two constraints that make the following boolean expressions true:

$$\forall w,t$$

$$\Big((\sum_{i=1}^{I} LSREADY_{w,i,t} = 0) \vee (\sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} = \sigma_L)\Big) \tag{22}$$

and

$$\forall w, t$$

$$\Big((\sum_{i=1}^{I} CCREADY_{w,i,t} = 0) \vee (\sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t} = \sigma_C)\Big) \tag{23}$$

Let us consider constraint (22). The equality

$$\sum_{i=1}^{I} LSREADY_{w,i,t} = 0 \tag{24}$$

holds iff $\forall i \quad LSREADY_{w,i,t} = 0$.

From the definition, we know that $LSREADY_{w,i,t} = 0$ iff the following boolean expressions hold:

$$\neg\Big((IL_1 = 1) \wedge (t < Y_{w,1})\Big) = true \tag{25}$$

for $LSREADY_{w,1,t} = 0$;

$$\neg\Big((Y_{w,i-1} < t) \wedge (IL_i = 1) \wedge (t < Y_{w,i})\Big) = true \tag{26}$$

for $LSREADY_{w,i,t} = 0 \quad \forall i = 2..I$.

Expressions (25) and (26) can be equivalently rewritten as:

$$(IL_1 = 0) \vee (t \geq Y_{w,1}) = true \tag{27}$$

for $LSREADY_{w,1,t} = 0$;

$$(Y_{w,i-1} \geq t) \vee (IL_i = 0) \vee (t \geq Y_{w,i}) = true \tag{28}$$

for $LSREADY_{w,i,t} = 0 \quad \forall i = 2..I$.

Taking into account that

$$\sum_{t'=1}^{t}(LS_{w,i,t'} + CC_{w,i,t'}) = \begin{cases} 1 & \text{if } t \geq Y_{w,i}; \\ 0 & \text{otherwise.} \end{cases}$$

and

$$\sum_{t'=t}^{T}(LS_{w,i,t'} + CC_{w,i,t'}) = \begin{cases} 1 & \text{if } Y_{w,i} \geq t; \\ 0 & \text{otherwise.} \end{cases}$$

we can rewrite the left hand sides of boolean expressions (27) and (28) as

$$TL_{w,1,t} = (IL_1 = 0) \vee (\sum_{t'=1}^{t}(LS_{w,1,t'} + CC_{w,1,t'}) = 1)$$

and

$$\begin{aligned} TL_{w,i,t} =& (\sum_{t'=t}^{T}(LS_{w,i-1,t'} + CC_{w,i-1,t'}) = 1) \vee (IL_i = 0) \vee \\ & (\sum_{t'=1}^{t}(LS_{w,i,t'} + CC_{w,i,t'}) = 1) \quad \forall i = 2..I \end{aligned}$$

respectively (using the shorthand $TL_{w,i,t}$ for the purpose of making equations more readable).

In such a way the equality (24) can be equivalently rewritten as:

$$TL_{w,1,t} \wedge TL_{w,2,t} \wedge \cdots \wedge TL_{w,I,t} = true \tag{29}$$

To express $\sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} = \sigma_L$, which is the right hand side part of (22), let us denote

$$E_t = \begin{cases} 1 & \text{if } \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} = \sigma_L \ ; \\ 0 & \text{otherwise.} \end{cases}$$

Intuitively, $E_t = 1$ iff there is no spare capacity of load/store units in the streaming multiprocessor at clock cycle t. An equivalent (but more convenient) definition of the above binary decision variable is:

$$E_t = 1 - sign(\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}) \tag{30}$$

where

$$sign(r) = \begin{cases} 1 & \text{for } r > 0; \\ 0 & \text{for } r = 0; \\ -1 & \text{for } r < 0. \end{cases}$$

Subject to (29) and the definition of E_t, (22) is rewritten as:

$$(TL_{w,1,t} \wedge TL_{w,2,t} \wedge \cdots \wedge TL_{w,I,t}) \vee E_t \tag{31}$$

or equivalently

$$(TL_{w,1,t} \vee E_t) \wedge (TL_{w,2,t} \vee E_t) \wedge \cdots \wedge (TL_{w,I,t} \vee E_t) \tag{32}$$

We expressed the work-conserving property for load/store units through the boolean expressions presented above. To ensure that these expressions hold, we have to model them using linear constraints. According to Theorem 6 (see Appendix), Equation (31)

can be represented by a single relatively long linear constraint:

$$\begin{aligned}\forall w,t \quad & \frac{1}{2}(-\frac{I-1}{I}+\frac{1}{I}\sum_{i=1}^{I}TL_{w,i,t}+E_t)-\frac{1}{2I}< \\ & (TL_{w,1,t}\wedge TL_{w,2,t}\wedge\cdots\wedge TL_{w,I,t})\vee E_t \leq \\ & \frac{1}{I}\sum_{i=1}^{I}TL_{w,i,t}+E_t \end{aligned} \tag{33}$$

wherein the boolean expression $(TL_{w,1,t}\wedge TL_{w,2,t}\wedge\cdots\wedge TL_{w,I,t})\vee E_t$ is treated as a binary integer. Similarly expression (32) could be represented by I relatively short linear constraints:

$$\forall w,i,t \quad \frac{1}{2}(TL_{w,i,t}+E_t)\leq TL_{w,i,t}\vee E_t\leq TL_{w,i,t}+E_t \tag{34}$$

Applying a similar approach to (23), using shorthand $TC_{w,1,t}$, where

$$TC_{w,1,t}=(IC_1=0)\vee(\sum_{t'=1}^{t}(LS_{w,1,t'}+CC_{w,1,t'})=1)$$

$$\begin{aligned}TC_{w,i,t}=&(\sum_{t'=t}^{T}(LS_{w,i-1,t'}+CC_{w,i-1,t'})=1)\vee(IC_i=0)\vee \\ &(\sum_{t'=1}^{t}(LS_{w,i,t'}+CC_{w,i,t'})=1)\quad \forall i=2..I\end{aligned}$$

and binary decision variable

$$G_t=1-sign(\sigma_C-\sum_{w'=1}^{W}\sum_{i=1}^{I}CC_{w',i,t}) \tag{35}$$

we can present (23) by a single long constraint:

$$\begin{aligned}&\forall w, t\\&\frac{1}{2}(-\frac{I-1}{I}+\frac{1}{I}\sum_{i=1}^{I}TC_{w,i,t}+G_t)-\frac{1}{2I}<\\&(TC_{w,1,t}\wedge TC_{w,2,t}\wedge\cdots\wedge TC_{w,I,t})\vee G_t\leq\\&\frac{1}{I}\sum_{i=1}^{I}TC_{w,i,t}+G_t\end{aligned}\tag{36}$$

or by I short linear constraints:

$$\forall w, i, t \quad \frac{1}{2}(TC_{w,i,t}+G_t)\leq TC_{w,i,t}\vee G_t\leq TC_{w,i,t}+G_t \tag{37}$$

At this point, let us focus on how to model decision variables E_t and G_t (which have non-linear definitions) as linear expressions. By inspecting Equations (30) and (35), we can notice that function $sign()$ takes only **integer non-negative** arguments there. In the case of E_t, it is because $\forall t \quad \sigma_L \geq \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}$ In particular:

if $\sigma_L = \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}$, then
$sign(\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}) = sign(0) = 0$;

if $\sigma_L > \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}$, then
$sign(\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}) = 1$.

Since there is no need to "implement" $sign()$ for negative arguments, we can model it as follows. Let us denote the shorthand $SL_t = sign(\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t})$. The constraint

$$SL_t \leq \sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} \tag{38}$$

states the first basic property of the function (that its value cannot be greater than its argument). The second fundamental property (that the value of the function denotes

maximize $\sum_{t=1}^{T}(t \cdot (LS_{W,I,t} + CC_{W,I,t}))$ subject to		
iterated variables	expression for constraint	number of constraints
$\forall t$	$\sum_{w=1}^{W}\sum_{i=1}^{I} LS_{w,i,t} \leq \sigma_L$	T
$\forall t$	$\sum_{w=1}^{W}\sum_{i=1}^{I} CC_{w,i,t} \leq \sigma_C$	T
$\forall w = 1..(W-1)$	$\sum_{t=1}^{T}(t \cdot (LS_{w,I,t} + CC_{w,I,t})) \leq \sum_{t=1}^{T}(t \cdot (LS_{W,I,t} + CC_{W,I,t}))$	$W-1$
$\forall w, t$	$\sum_{i=1}^{I} LS_{w,i,t} \leq 1$	$W \cdot T$
$\forall w, t$	$\sum_{i=1}^{I} CC_{w,i,t} \leq 1$	$W \cdot T$
$\forall w, i$	$\sum_{t=1}^{T} LS_{w,i,t} = IL_i$	$W \cdot I$
$\forall w, i$	$\sum_{t=1}^{T} CC_{w,i,t} = IC_i$	$W \cdot I$
$\forall w, i = 1..(I-1)$	$1 + \sum_{t=1}^{T}(t \cdot (LS_{w,i,t} + CC_{w,i,t})) \leq \sum_{t=1}^{T}(t \cdot (LS_{w,i+1,t} + CC_{w,i+1,t}))$	$W \cdot (I-1)$
$\forall t$	$E_t \geq 1 - \sigma_L + \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}$	T
$\forall t$	$E_t \cdot \sigma_L \leq \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}$	T
$\forall w, i, t$	$\frac{1}{2}(TL_{w,i,t} + E_t) \leq TL_{w,i,t} \vee E_t \leq TL_{w,i,t} + E_t$	$W \cdot I \cdot T$
$\forall t$	$G_t \geq 1 - \sigma_C + \sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t}$	T
$\forall t$	$G_t \cdot \sigma_C \leq \sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t}$	T
$\forall w, i, t$	$\frac{1}{2}(TC_{w,i,t} + G_t) \leq TC_{w,i,t} \vee G_t \leq TC_{w,i,t} + G_t$	$W \cdot I \cdot T$

Figure 9: The complete ILP formulation (using short constraints)

the sign of the argument) is stated by the following constraint:

$$\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} \leq SL_t \cdot (\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t})$$

Without loss of correctness, we can rewrite this as

$$\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} \leq SL_t \cdot \sigma_L \tag{39}$$

reducing computational complexity for the software implementation.

According to Equation (30) and the definition of SL_t, we can compute E_t as $(1 - SL_t)$, but it will be more efficient to model E_t directly (using the previous

derivation of SL_t). Multiplying (38) by (-1) and adding 1 to both sides yields

$$1 - SL_t \geq 1 - (\sigma_L - \sum_{w'=1}^{W} \sum_{i=1}^{I} LS_{w',i,t})$$

This can be rewritten as follows:

$$E_t \geq 1 - \sigma_L + \sum_{w'=1}^{W} \sum_{i=1}^{I} LS_{w',i,t} \tag{40}$$

Multiplying (39) by (-1) we get

$$-(\sigma_L - \sum_{w'=1}^{W} \sum_{i=1}^{I} LS_{w',i,t}) \geq (1 - SL_t) \cdot \sigma_L - \sigma_L$$

One may then reduce it to

$$E_t \cdot \sigma_L \leq \sum_{w'=1}^{W} \sum_{i=1}^{I} LS_{w',i,t} \tag{41}$$

By analogy, for linear constraints (40) and (41) for G_t:

$$G_t \geq 1 - \sigma_C + \sum_{w'=1}^{W} \sum_{i=1}^{I} CC_{w',i,t}$$

$$G_t \cdot \sigma_C \leq \sum_{w'=1}^{W} \sum_{i=1}^{I} CC_{w',i,t}$$

5.3 Alternative optimization problem formulation

The number of decision variables, the expression to use as an objective function, the number of constraints and their complexity — all these characteristics of the formulation affect the computational time for solving the optimization problem. Instead of using binary decision variables for every type of computational unit (e.g., binary variables $CC_{w,i,t}$ for CUDA cores) we introduce more generic integer variables

$Y_{w,i} \in \{1, \dots T\}$, where $(Y_{w,i} = t)$ denotes that warp w performs instruction i at clock cycle t. These variables were already utilized in Section 5.2.3 as auxiliary variables just for the sake of presenting the derivation. Here, we would like to use $Y_{w,i}$ as integer decision variables and reformulate the optimization problem according to that. Although $Y_{w,i}$ does not explicitly specify the type of the computational unit that is performing the instruction for the warp w, we can always find it out with the help of the instruction index i and binary constants IL_i, IC_i, etc.

To simplify the expression of the objective function we utilize the idea presented in Section 5.2.1 enforcing the warp with identifier W to be the last one to finish execution of the kernel instruction string. This can be stated with the help of $W - 1$ linear constraints:

$$\forall w \in \{1, \dots (W-1)\} \quad Y_{w,I} \leq Y_{W,I}$$

Now we are sure that the later that warp W performs the instruction with index I, the longer the makespan we will get. Therefore, our objective function is

$$Maximize \quad Y_{W,I}$$

Without the loss of generality, the solution search space can be reduced by restricting the finishing time of the warps. For instance, we can optionally order the warps according to their index, such that, the warp with the higher index finishes its execution no earlier when compared to the warp with the lower index. This statement can be expressed as the following $W - 1$ constraints:

$$\forall w \in \{1, \dots (W-1)\} \quad Y_{w,I} \leq Y_{w+1,I} \tag{42}$$

Although, the constraints in Equation (42) are optional, in our implementation of the ILP, they contributed to the speed-up. However, in general case, the usefulness of

these constraints is subject to particular ILP-solver organization, amount of memory available, etc.

Every warp should perform the instructions of the kernel instruction string in a given order (instruction i should be executed before instruction $i+1$). Hence, we need $W \cdot (I-1)$ precedence constraints:

$$\forall w, i \in \{1 \dots (I-1)\} \quad Y_{w,i} < Y_{w,i+1}$$

The possible scenario, when warp w performs instruction i at clock cycle t ($Y_{w,i} = t$), can be expressed in the following way:

$$(Y_{w,i} \leq t) \wedge (t \leq Y_{w,i}) \tag{43}$$

We use Equation (43) and binary constants IL_i, IC_i, etc., to ensure that the capacity of computational units of a streaming multiprocessor is not exceeded at any clock cycle.

With regard to load/store units the idea behind the corresponding capacity constraints is the following: at every clock cycle no more than σ_L "L"-instructions can be performed. This statement can be expressed as T constraints:

$$\forall t \in \{1, \dots T\} \quad \sum_{w=1}^{W} \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) \wedge (t \leq Y_{w,i}) \wedge (IL_i = 1) \Big) \leq \sigma_L \tag{44}$$

Let us consider the three terms $(Y_{w,i} \leq t)$, $(t \leq Y_{w,i})$, $(IL_i = 1)$ that form a body of summation in the left-hand side of Equation (44). We can treat these terms as binary expressions and binary constants, such that:

$$(Y_{w,i} \leq t) = \begin{cases} 1 & \text{if} \quad Y_{w,i} \leq t; \\ 0 & \text{otherwise.} \end{cases} \tag{45}$$

$$(t \leq Y_{w,i}) = \begin{cases} 1 & \text{if} \quad t \leq Y_{w,i}; \\ 0 & \text{otherwise.} \end{cases} \tag{46}$$

$$(IL_i = 1) = IL_i \tag{47}$$

Thus, we can consider the whole term $\Big((Y_{w,i} \leq t) \wedge (t \leq Y_{w,i}) \wedge (IL_i = 1)\Big)$ as a conjunction of three binary expressions. Therefore, we can apply Theorem 5 (see Appendix) as follows: the entire right-hand side of Equation (44) maps to X. The individual terms $(Y_{w,i} \leq t),\ (t \leq Y_{w,i}),\ (IL_i = 1)$ that form the conjunction map to $I = 3$ terms $x_1,\ x_2,\ x_3$. Therefore,

$$\sum_{w=1}^{W} \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) \wedge (t \leq Y_{w,i}) \wedge (IL_i = 1)\Big)$$

is equivalent to

$$\begin{gathered} \forall \quad w \in \{1, \dots W\} \quad i \in \{1, \dots I\} : \\ -\frac{2}{3} + \frac{1}{3}\Big((Y_{w,i} \leq t) + (t \leq Y_{w,i}) + (IL_i = 1)\Big) \leq \\ \Big((Y_{w,i} \leq t) \wedge (t \leq Y_{w,i}) \wedge (IL_i = 1)\Big) \leq \\ \frac{1}{3}\Big((Y_{w,i} \leq t) + (t \leq Y_{w,i}) + (IL_i = 1)\Big) \end{gathered} \tag{48}$$

Equation (48) represents $W \times I$ inequalities for every clock cycle t. By summing all of them we get the following constraint:

$$\forall t \in \{1, \dots T\}$$

$$-\frac{2 \times W \times I}{3} + \frac{1}{3} \sum_{w=1}^{W} \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) + (t \leq Y_{w,i}) + (IL_i = 1) \Big) \leq \sum_{w=1}^{W} \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) \wedge (t \leq Y_{w,i}) \wedge (IL_i = 1) \Big) \leq \frac{1}{3} \sum_{w=1}^{W} \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) + (t \leq Y_{w,i}) + (IL_i = 1) \Big) \tag{49}$$

Notice, the middle component of the double inequality in Equation (49) is equal to the left-hand side of the capacity constraint in Equation (44). Therefore, with the help of the constraints stated in Equation (44) and in Equation (49), the capacity requirements for load/store units of a streaming multiprocessor can be expressed in a linear way.

The very same reasoning can be applied for the derivation of the capacity constraints for other types of computational units. Analogously to Equation (44) and to Equation (49), the capacity constraints for CUDA cores can be expressed as follows:

$$\forall t \in \{1, \dots T\} \quad \sum_{w=1}^{W} \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) \wedge (t \leq Y_{w,i}) \wedge (IC_i = 1) \Big) \leq \sigma_C \tag{50}$$

$$\forall t \in \{1, \dots T\}$$

$$-\frac{2 \times W \times I}{3} + \frac{1}{3} \sum_{w=1}^{W} \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) + (t \leq Y_{w,i}) + (IC_i = 1) \Big) \leq \sum_{w=1}^{W} \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) \wedge (t \leq Y_{w,i}) \wedge (IC_i = 1) \Big) \leq \frac{1}{3} \sum_{w=1}^{W} \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) + (t \leq Y_{w,i}) + (IC_i = 1) \Big) \tag{51}$$

One may notice that boolean expressions $(Y_{w,i} \leq t)$ and $(t \leq Y_{w,i})$ used in Equation (49) and in Equation (51) are actually not linear, but rather conditional statements represented in Equation (45) and in Equation (46) respectively. However, we intentionally presented all the derivations in Equations (44)– (51) using these boolean expressions because of the following reason. Since linear programming is a popular tool in many domains including management decision support, the vendors of solvers and development environments provide users with high-level abstraction languages to express their optimization problems in a form usually called "models". The languages of this kind e.g., OPL [95], allow to formulate models in a significantly easier and informal way when compared to pure linear programming formalism. In such cases, the workload of translating the model to a proper linear program is taken by the development environment. Of course, this is done at a price of twofold waste of performance:

- by translating the model offline;
- by executing an automatically generated program that is potentially less efficient when compared to a handmade linear program.

However, in terms of presentation, high-level models are favorable, since they require less effort from the reader to follow the derivations. Therefore, we opt to use boolean expressions $(Y_{w,i} \leq t)$ and $(t \leq Y_{w,i})$ in the following derivations, although here we will show how to express linearly the conditional statements represented in Equation (45) and in Equation (46).

Let us consider boolean expression $(Y_{w,i} \leq t)$ first. Based on the corresponding conditional statement (see Equation (45)) we can introduce a boolean variable $B_{w,i,t}$ which states whether the warp w executes instruction i **before than** or exactly at

the clock cycle t:

$$\forall \quad w \in \{1, \dots W\} \quad i \in \{1, \dots I\} \quad t \in \{1, \dots T\} :$$
$$B_{w,i,t} = \begin{cases} 1 & \text{if} \quad Y_{w,i} \leq t; \\ 0 & \text{otherwise.} \end{cases} \tag{52}$$

In the following reasoning we rely on the pessimistic makespan estimate T that was obtained using the technique presented in Section 5.1, thus, by the definitions of t, and $Y_{w,i}$ we know that

$$1 \leq t \leq T$$
$$1 \leq Y_{w,i} \leq T \tag{53}$$

To ensure that $B_{w,i,t}$ takes appropriate values consistent with Equation (52), we can construct the following linear constraints:

$$\forall \quad w \in \{1, \dots W\} \quad i \in \{1, \dots I\} \quad t \in \{1, \dots T\}$$
$$\begin{cases} t - Y_{w,i} < B_{w,i,t} \times T \\ t - Y_{w,i} \geq (B_{w,i,t} - 1) \times T \end{cases} \tag{54}$$

The validity of the constraints expressed above can be checked by mapping possible values of binary variable $B_{w,i,t}$ into Equation (54) and checking against the definition

in Equation (52) and properties expressed by Equation (53):

$$\forall \quad w \in \{1, \dots W\} \quad i \in \{1, \dots I\} \quad t \in \{1, \dots T\}$$

$$Case \quad B_{w,i,t} = 0 : \overset{(54)}{\Longrightarrow} \begin{cases} t < Y_{w,i} \\ t - Y_{w,i} \geq -T \end{cases} \tag{55}$$

Notice, that $(t < Y_{w,i})$ complies with the definition of $B_{w,i,t}$ for this case and $(t - Y_{w,i} \geq -T)$ is just a valid inequality (from Equation (53)).

$$Case \quad B_{w,i,t} = 1 \overset{(54)}{\Longrightarrow} \begin{cases} t - Y_{w,i} < T \\ Y_{w,i} \leq t \end{cases} \tag{56}$$

In Equation (56), $(t - Y_{w,i} < T)$ is a valid inequality (from Equation (52) and Equation (53)), while $(Y_{w,i} \leq t)$ corresponds to the definition of $B_{w,i,t}$, for this case.

For the boolean expression $(t \leq Y_{w,i})$ and conditional statement in Equation (46) we apply similar reasoning as we did for the boolean expression $(Y_{w,i} \leq t)$ and the conditional statement in Equation (45). We introduce a boolean variable $A_{w,i,t}$ which specifies whether the warp w executes instruction i **after than** or exactly at the clock cycle t:

$$\forall \quad w \in \{1, \dots W\} \quad i \in \{1, \dots I\} \quad t \in \{1, \dots T\}$$

$$A_{w,i,t} = \begin{cases} 1 & \text{if} \quad t \leq Y_{w,i}; \\ 0 & \text{otherwise.} \end{cases} \tag{57}$$

The compliance of the values of $A_{w,i,t}$ with Equation (57) is guaranteed by the

following linear constraints:

$$\forall \quad w \in \{1, \dots W\} \quad i \in \{1, \dots I\} \quad t \in \{1, \dots T\}$$
$$\begin{cases} Y_{w,i} - t < A_{w,i,t} \times T \\ Y_{w,i} - t \geq (A_{w,i,t} - 1) \times T \end{cases} \tag{58}$$

Let us demonstrate, how the constraints in Equation (58) determine the values of $A_{w,i,t}$. Similarly to what we did in Equation (55) and in Equation (56), we just map all possible values of $A_{w,i,t}$ into linear constraints in Equation (58).

$$Case \quad A_{w,i,t} = 0$$
$$\begin{cases} Y_{w,i} < t \\ Y_{w,i} - t \geq -T \end{cases} \tag{59}$$

While $(Y_{w,i} < t)$ corresponds to the definition of $A_{w,i,t}$ in Equation (57), the inequality $(Y_{w,i} - t \geq -T)$ is simply valid (from Equation (53)).

$$Case \quad A_{w,i,t} = 1$$
$$\begin{cases} Y_{w,i} - t < T \\ Y_{w,i} \geq t \end{cases} \tag{60}$$

In Equation (60), expression $(Y_{w,i} - t < T)$ is a valid inequality (from Equation (53)) and $(Y_{w,i} \geq t)$ complies with the conditional statement in Equation (57).

After including constraints represented in Equation (58) and in Equation (54) into the ILP formulation we can express other constraints in a purely linear way. Let us consider the load/store units capacity constraint in Equation (48). By substituting $(Y_{w,i} \leq t)$ and $(t \leq Y_{w,i})$ by $B_{w,i,t}$ and $A_{w,i,t}$ respectively, the constraint

in Equation (48) is equivalent to the following constraint:

$$\begin{aligned} \forall \quad w \in \{1, \dots W\} \quad i \in \{1, \dots I\} : \\ -\frac{2}{3} + \frac{1}{3}\Big(B_{w,i,t} + A_{w,i,t} + IL_i\Big) \leq \\ \Big(B_{w,i,t} \wedge A_{w,i,t} \wedge IL_i\Big) \leq \\ \frac{1}{3}\Big(B_{w,i,t} + A_{w,i,t} + IL_i\Big) \end{aligned} \tag{61}$$

Applying the very same substitution to all the constraints in this section would allow us to make our ILP truly linear. Still, we need to discuss under which cases one or the other ILP formulation would be preferable.

We showed that the integer variable $Y_{w,i}$ can be considered as a "decision variable" only from the viewpoint of the user that formulates the ILP in a modern development environment (e.g., IBM CPLEX [95]). While the user can stay on the higher level of abstraction, the development environment will eventually translate the model to the integer linear program in terms of pure decision variables $A_{w,i,t}$ and $B_{w,i,t}$. Then the linear program will be transferred to the ILP solver which will be able to provide the solution (in case the program is feasible) in the form of particular values for binary decision variables $A_{w,i,t}$ and $B_{w,i,t}$. After that, the development environment will translate that solution to a representation via the values of variables $Y_{w,i}$. Thus, the simplicity provided to the user comes at a price of additional workload delegated to development environment. Therefore, for rapid prototyping, the usage of variable $Y_{w,i}$ as a high-level pseudo "decision variable" in an ILP model would be appropriate. While in the case when performance is crucial, one should consider formulating the integer linear program directly in terms of binary decision variables $A_{w,i,t}$ and $B_{w,i,t}$, to be processed by the solver.

Also, we need to discuss what are the other benefits and potential drawbacks of the ILP formulation presented in this section when compared to the original one presented

in Section 5.2. The reader may notice that, after all the derivations involving two-dimensional variable $Y_{w,i}$ we come up with an ILP formulation that eventually would need to be based once again on two three-dimensional binary decision variables $A_{w,i,t}$ and $B_{w,i,t}$ even though these are used in implicit way. What are the benefits of this formulation? How does this formulation differ from the one presented in Section 5.2 using three-dimensional binary decision variables $LS_{w,i,t}$ and $CC_{w,i,t}$?

In Section 5.2 we proposed to use a distinct variable for every type of computational unit within a streaming multiprocessor – e.g., $LS_{w,i,t}$ for the load/store units, etc. However, the formulation presented in the current section uses a more generic approach in terms of the defining decision variables. Since variable $Y_{w,i}$ does not explicitly hold the information about the type of computational unit on which the warp w has to execute the instruction i, we rely on the binary constants that are defined for every type of computational unit, e.g., IL_i etc. According to the assumptions listed in Section 4.3, in this chapter we consider the case when a streaming multiprocessor includes only two types of computational units: load/store units and CUDA cores. Therefore, for such a case there is no expectation to gain performance with this generalization because of the following reasons:

- The solver would still need two three-dimensional binary decision variables $A_{w,i,t}$ and $B_{w,i,t}$ for the variable $Y_{w,i}$ to be expressed in the integer linear program;

- The workload of determining the proper unit for the instruction i still has to be specified in the constraints rather than in the decision variable formulation.

However, for more realistic models of a streaming multiprocessor that would also include other types of computational units e.g., special function units, double precision units, etc., the use of just two generic variables $A_{w,i,t}$ and $B_{w,i,t}$ could be beneficial.

Since any warp is able to perform at most a single instruction at any given clock

cycle, we need $W \times T$ constraints

$$\forall w, t \quad \sum_{i=1}^{I} \Big((Y_{w,i} \leq t) \wedge (t \leq Y_{w,i}) \Big) \leq 1 \tag{62}$$

In the case of integer decision variables, the work-conserving property of the scheduler can be expressed in a similar way as it was done for binary decision variables in Section 5.2.4. However, some steps have to be represented here for the sake of clarity. In Section 5.2.4, we introduced the work-conserving constraints with the help of auxiliary variables. Then we showed how the high-level formulation (that uses auxiliary variables) can be transformed to a low-level formulation (that uses actual decision variables). Here, we would like to follow the same formulation strategy. Moreover, we rely on the same auxiliary variables as we did in Section 5.2.4.

$$\forall w \in \{1, \dots W\}, i \in \{1, \dots I\}, t \in \{1, \dots T\} : \quad LSREADY_{w,i,t}, \quad CCREADY_{w,i,t}$$

We showed that the work-conserving property formulated in Equation (21) with the help of these auxiliary variables, can be split into shorter constraints for every type of computational unit, e.g., for the load/store units in Equation (22) and for the CUDA cores in Equation (23). Let us consider the work-conserving constraints for load/store units in Equation (22):

$$\forall w, t \quad \Big((\sum_{i=1}^{I} LSREADY_{w,i,t} = 0) \vee (\sum_{w'=1}^{W} \sum_{i=1}^{I} LS_{w',i,t} = \sigma_L) \Big)$$

These constraints state that there should be **either** no idle warps with pending load/store workload **or** if such warps exist, it should be due to lack of spare capacity in terms of load/store units in a streaming multiprocessor; thus, not all ready warps will be able to perform the computation. These two alternatives are represented by the left-hand expression and the right-hand expression of the disjunction

stated in Equation (22) respectively.

Let us consider the left-hand expression of the disjunction in Equation (22):

$$\sum_{i=1}^{I} LSREADY_{w,i,t} = 0$$

This equation can be rewritten as

$$LSREADY_{w,1,t} + \sum_{i=2}^{I} LSREADY_{w,i,t} = 0 \tag{63}$$

As we already showed in Equation (25), $(LSREADY_{w,1,t} = 0)$ corresponds to

$$\Big((IL_1 = 0) \vee (t \geq Y_{w,1}) = true\Big)$$

and $\forall i \in \{2..I\}$ $(LSREADY_{w,i,t} = 0)$ corresponds to (Equation (26))

$$\Big((Y_{w,i-1} \geq t) \vee (IL_i = 0) \vee (t \geq Y_{w,i}) = true\Big)$$

Let us rewrite Equation (25) by substituting $(IL_1 = 0)$ and $(t \geq Y_{w,1})$ by $\neg IL_1$ and $B_{w,1,t}$ respectively:

$$(\neg IL_1) \vee B_{w,1,t} = true \tag{64}$$

Similarly, Equation (26) can be rewritten by substituting $(Y_{w,i-1} \geq t)$, $(IL_i = 0)$ and $(t \geq Y_{w,i})$ by $A_{w,i-1,t}$, $\neg IL_i$ and $B_{w,i,t}$:

$$A_{w,i-1,t} \vee (\neg IL_i) \vee B_{w,i,t} = true \tag{65}$$

Thus, $(LSREADY_{w,1,t} = 0)$ corresponds to Equation (64) and
$\forall i = 2 \ldots I$ $(LSREADY_{w,i,t} = 0)$ corresponds to Equation (65).

One may notice that Equation (63) is equivalent to the following condition

$$\forall \quad i \in \{1, \dots I\} \quad LSREADY_{w,i,t} = 0 \tag{66}$$

Therefore, Equation (63) can be expressed in a following way:

$$(LSREADY_{w,1,t} = 0) \wedge (LSREADY_{w,2,t} = 0) \wedge \dots \wedge (LSREADY_{w,I,t} = 0) \tag{67}$$

We can rewrite Equation (67) using Equation (64) and Equation (65):

$$(\neg IL_1 \vee B_{w,1,t}) \wedge (A_{w,1,t} \vee \neg IL_2 \vee B_{w,2,t}) \wedge \dots \wedge (A_{w,I-1,t} \vee \neg IL_I \vee B_{w,I,t}) = true \tag{68}$$

Equation (68) is just another way of representing the left-hand expression of the disjunction in Equation (22) with the help of binary decision variables $A_{w,i,t}$ and $B_{w,i,t}$. Notice, that Equation (68) is based on the idea of representing Equation (22) with the help of boolean expressions, just like we did for the previous ILP formulation to obtain Equation (29) in Section 5.2.4.

Let us consider the right-hand expression of the disjunction in Equation (22)

$$(\sum_{w'=1}^{W} \sum_{i=1}^{I} LS_{w',i,t} = \sigma_L) \tag{69}$$

When this equation holds, it means that for any clock cycle t, the capacity of load/store units in a given streaming multiprocessor is fully utilized. This is done by considering the value of the $LS_{w',i,t}$ variable for any warp $w' \in \{1..W\}$ and any instruction $i \in \{1..I\}$ from the kernel instruction string. Since in this section we are trying to get rid of "three-dimensional" decision variables $LS_{w',i,t}$, $CC_{w',i,t}$ etc., we need to express Equation (69) by using integer decision variables $Y_{w',i}$ and binary constants IL_i. According to the definition stated in Section 5.2, the variable $LS_{w',i,t}$

specifies whether the warp w' executes instruction i on a load/store unit at the clock cycle t. This can be expressed as a conjunction of the following two statements:

- The warp w' performs an instruction i at the clock cycle t;
- An instruction i is for a load/store unit.

The first of these two statements can be expressed with the help of Equation (43)

$$(Y_{w',i} \leq t) \wedge (t \leq Y_{w',i})$$

The second statement can be expressed by considering the value of the variable IL_i that was defined in Section 5.2

$$(IL_i = 1)$$

Hence, $LS_{w',i,t}$ is equal to

$$\Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1)\Big)$$

and Equation (69) corresponds to the following equation:

$$\sum_{w'=1}^{W} \sum_{i=1}^{I} \Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1)\Big) = \sigma_L \tag{70}$$

Equation (70) has to be expressed in a linear way. Similarly to what we did in Section 5.2.4, we use an auxiliary variable

$$E_t = \begin{cases} 1 & \text{if } \sum_{w'=1}^{W} \sum_{i=1}^{I} \Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1)\Big) = \sigma_L \ ; \\ 0 & \text{otherwise.} \end{cases} \tag{71}$$

Thus, E_t is a binary auxiliary variable that is equal to 1 if at clock cycle t there is *no* spare capacity of load/store units in a streaming multiprocessor. We can express

this variable in a more convenient way as follows:

$$E_t = 1 - sign(\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I}\Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1)\Big)) \tag{72}$$

The function $sign()$ in Equation (72) has to be modeled in a linear way. It is important to notice that its argument

$$\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I}\Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1)\Big)$$

is non-negative because of the load/store capacity constraint in Equation (44). This allows us to model $sign()$ in a simpler and more efficient way, as it was shown in Section 5.2.4. The idea behind this modeling is based on the following observation. When being a function of an **integer non-negative argument**, $sign()$ has two fundamental properties:

- The value of the function cannot be greater than its argument.
- The value of the function specifies the sign of the argument.

Similarly as we did in Section 5.2.4, we are going to use a shorthand

$$SL_t = sign(\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I}\Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1)\Big)$$

to represent the value of the function $sign()$ in Equation (72) where it takes the following argument:

$$\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I}\Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1)\Big)$$

The first basic property can be expressed as follows:

$$SL_t \leq \sigma_L - \sum_{w'=1}^{W} \sum_{i=1}^{I} \Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1) \Big) \tag{73}$$

The second basic property is stated with the help of the following equation:

$$\sigma_L - \sum_{w'=1}^{W} \sum_{i=1}^{I} \Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1) \Big) \leq SL_t \cdot \sigma_L \tag{74}$$

The linear modeling of $sign()$ presented above is based on an implicit assumption that the expression

$$\sum_{w'=1}^{W} \sum_{i=1}^{I} \Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1) \Big) \tag{75}$$

can be modeled in a linear way as well. We have already showed how to do it for the analogous boolean expression in Equation(49). Therefore, for the argument presented in Equation (75), linear constrains can be written as follows:

$$\begin{aligned} & \forall t \in \{1, \dots T\} \\ -\frac{2 \times W \times I}{3} + \frac{1}{3} \sum_{w'=1}^{W} \sum_{i=1}^{I} \Big((Y_{w',i} \leq t) + (t \leq Y_{w',i}) + (IL_i = 1) \Big) \leq \\ \sum_{w'=1}^{W} \sum_{i=1}^{I} \Big((Y_{w',i} \leq t) \wedge (t \leq Y_{w',i}) \wedge (IL_i = 1) \Big) \leq \\ \frac{1}{3} \sum_{w'=1}^{W} \sum_{i=1}^{I} \Big((Y_{w',i} \leq t) + (t \leq Y_{w',i}) + (IL_i = 1) \Big) \end{aligned} \tag{76}$$

The left-hand expression of the disjunction used in a load/store work-conserving requirement (Equation (22)) corresponds to Equation (68) and the right-hand expression of that disjunction corresponds to Equation (70). From the definition of the binary auxiliary variable E_t presented in Equation (71), the load/store full capac-

ity requirement represented in Equation (70) is equivalent to the boolean expression $(E_t = 1)$. Therefore, the complete work-conserving requirement for the load/store units in Equation (22) can be expressed with the help of Equation (68) and E_t as follows:

$$\begin{aligned} &\forall \quad w \in \{1, \dots W\} \quad t \in \{1, \dots T\} \\ &\Big((\neg IL_1 \vee B_{w,1,t}) \wedge \dots \wedge (A_{w,I-1,t} \vee \neg IL_I \vee B_{w,I,t})\Big) \vee E_t = true \end{aligned} \tag{77}$$

Equation (77) can be equivalently rewritten as

$$\begin{aligned} &\forall \quad w \in \{1, \dots W\} \quad t \in \{1, \dots T\} \\ &(\neg IL_1 \vee B_{w,1,t} \vee E_t) \wedge \dots \wedge (A_{w,I-1,t} \vee \neg IL_I \vee B_{w,I,t} \vee E_t) = true \end{aligned} \tag{78}$$

Similarly to Equation (32), Equation (78) can be presented as a list of I relatively short equations, which must simultaneously hold *true*:

$$\begin{aligned} (\neg IL_1) \vee B_{w,1,t} \vee E_t &= true \\ A_{w,1,t} \vee (\neg IL_2) \vee B_{w,2,t} \vee E_t &= true \\ &\dots \\ A_{w,I-1,t} \vee (\neg IL_I) \vee B_{w,I,t} \vee E_t &= true \end{aligned} \tag{79}$$

To express these equations in a linear way we can use Theorem 4 (see Appendix). For the first equation from the list (79)

$$(\neg IL_1) \vee B_{w,1,t} \vee E_t = true$$

we map the whole left-hand side of that equation to X, and $(I = 3)$ operands of the

disjunction operation, namely $(1 - IL_1),\ B_{w,i,t},\ E_t$ we map to $x_1,\ x_2,\ x_3$ respectively.

$$\begin{aligned} \frac{1}{3} \times \Big((1 - IL_1) + B_{w,1,t} + E_t\Big) \leq \\ (\neg IL_I) \vee B_{w,1,t} \vee E_t) \leq \\ \Big((1 - IL_1) + B_{w,1,t} + E_t\Big) \end{aligned} \tag{80}$$

For every single equation from the list (79) except the first one,

$$\forall i \in \{2 \dots I\} \quad A_{w,i-1,t} \vee (\neg IL_i) \vee B_{w,i,t} \vee E_t = true$$

we map the whole left-hand side of that equation to X, and $(I = 4)$ operands of the disjunction operation, namely $A_{w,i,t},\ (1 - IL_i),\ B_{w,i,t},\ E_t$ we map to $x_1,\ x_2,\ x_3,\ x_4$ respectively.

$$\begin{aligned} \frac{1}{4} \times \Big(A_{w,i-1,t} + (1 - IL_i) + B_{w,i,t} + E_t\Big) \leq \\ A_{w,i-1,t} \vee (\neg IL_i) \vee B_{w,i,t} \vee E_t) \leq \\ \Big(A_{w,i-1,t} + (1 - IL_i) + B_{w,i,t} + E_t\Big) \end{aligned} \tag{81}$$

Therefore, from Equation (80) and Equation (81), the work-conserving property of the scheduler can be expressed with the help of $W \times I \times T$ relatively short linear

constraints as follows:

$$\begin{aligned}
\forall \quad w \in \{1, \dots W\} \quad t \in \{1, \dots T\} \\
& \frac{1}{3} \times \Big((1 - IL_1) + B_{w,1,t} + E_t \Big) \leq \\
& (\neg IL_I) \vee B_{w,1,t} \vee E_t) \leq \\
& \Big((1 - IL_1) + B_{w,1,t} + E_t \Big) \\
\forall i \in \{2, \dots I\} \\
& \frac{1}{4} \times \Big(A_{w,i-1,t} + (1 - IL_i) + B_{w,i,t} + E_t \Big) \leq \\
& A_{w,i-1,t} \vee (\neg IL_i) \vee B_{w,i,t} \vee E_t) \leq \\
& \Big(A_{w,i-1,t} + (1 - IL_i) + B_{w,i,t} + E_t \Big) \qquad (82)
\end{aligned}$$

We consider the ILP-formulation derived in this section to be more generic and suitable for the models of GPU architectures with greater variety of computational units incorporated. However, in the context of the simplified model powered by CUDA cores and load/store units only considered in this thesis, the ILP-formulation derived in Section 5.2 seems to be the best fit.

5.4 Summary of the ILP formulation

Let us now present the entire formulation of the binary ILP from Section 5.2 in one place (Figure 9) (opting for using as short constraints as possible).

5.5 Resolving the issue of tractability

Integer programming is in common use in various fields [184] and corresponding problems are probably, the most widely-used examples of NP-hard computational problems. Even for relatively small number of warps (W), computing the makespan with the ILP formulation presented above may take much time. However, we can find a

marginally pessimistic makespan estimate at only a fraction of the time as follows:

Let $T^{(W)}$ denote the worst-case makespan of W warps (for which we seek an upper bound) and $T^{(x)}$ denote the corresponding worst-case makespan for x warps (with $x \ll W$). By choosing a small enough value for x such that the exact value for $T^{(x)}$ can be tractably computed, according to our ILP derivation, then $T^{(W)}$ can be safely approximated by

$$T^{(W)}_{approx(x)} = \min_{y=1}^{x} T^{(W,y)} \tag{83}$$

where

$$T^{(W,y)} = \left\lceil \frac{W}{y} \right\rceil \cdot T^{(y)}, \quad y \in \mathbb{N} \tag{84}$$

We compute the upper bound on $T^{(W)}$ using Equation (83) and not as $T^{(W,x)}$ because, although $T^{(W,x)}$ typically decreases with increasing x, sometimes there are small increases (especially for very small x). The estimate $T^{(W)}_{approx(x)}$ for $T^{(W)}$ given by (83) improves with higher x, at the cost of rapidly increasing computation times. However, experimental evidence (see the next section) shows diminishing returns, even past small values of x. In other words, the estimate rapidly converges and even for small x, there is very little pessimism.

5.6 Experiments

As shown in Section 5.2 the makespan depends on the number of warps, the kernel instruction string and the hardware (namely, the number of computational units of each type and the warp size). We implemented the techniques introduced in above in a cross-platform software tool that reads the problem instance from a configuration file (Figure 10), constructs the binary ILP-formulation and launches the proprietary ILP-solver (see [95]). After getting the solution, it presents the worst-case makespan and corresponding schedule (like the one in Figure 8) or alternatively computes an

estimate using Equation (84) (if the user does not want to wait too long).

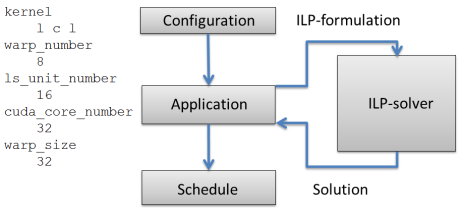


Figure 10: Typical configuration file and application workflow.

In Section 5.2.4 we stated that there are two alternatives for expressing the work-conserving property: either (i) using $W \cdot I \cdot T$ shorter constraints (34), (37) or (ii) using $W \cdot T$ longer constraints (33), (36). We implemented both options and compared their timings. One such comparison is presented in Figure 11. In our experiments, the first option generally gave shorter computation times.

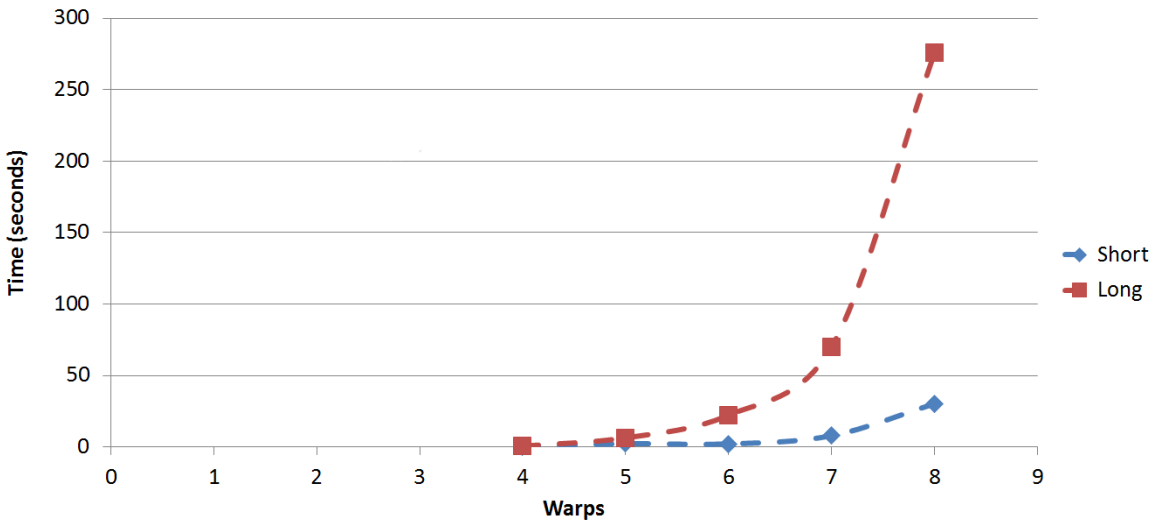


Figure 11: Computation time for solving ILP-problem with short and long constraints ($\sigma_L = \sigma_C = 1$, "LLCLL")

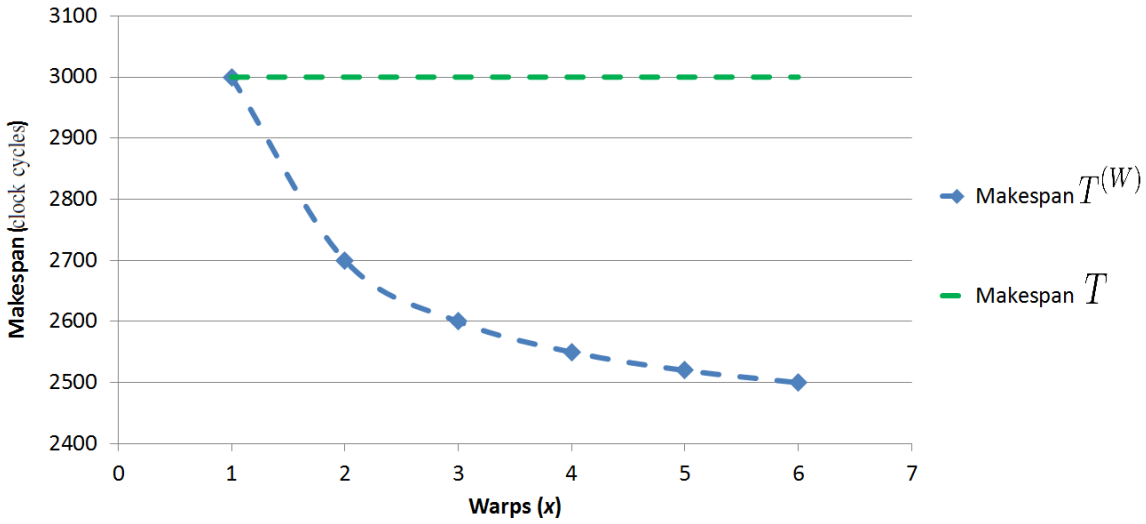


Figure 12: Convergence of $T^{(W)}$ with increasing x (W=600, σ_L=σ_C=1, "LLCLL"). The horizontal dashed line corresponds to the pessimistic estimate T (Section 5.1).

We also explored how the tractable approximation for $T^{(W)}$ (presented in Section 5.5) improves/converges with increased values of the parameter x. Figure 12 and Figure 13 present the results of two such experiments. In general, we observed that the estimate $T^{(W)}$ converges very fast with increasing x and afterwards the improvement to the estimate is minor (diminishing returns). Our interpretation is that this is because the approximation is good even for small values of x. Therefore, although the computation time increases very rapidly with x (Figure 13), one may obtain (i.e., using small x) estimates that are both quite accurate *and* tractably derivable.

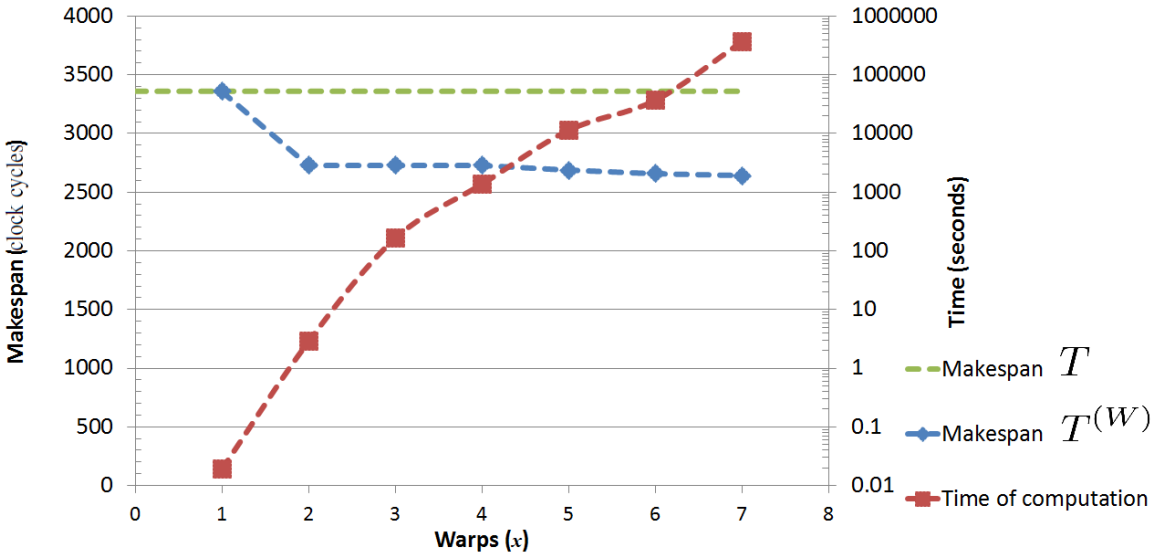


Figure 13: Growth of computation time and convergence of $T^{(W,x)}$ with increasing x ($W = 420$, $\sigma_L = \frac{1}{2}$, $\sigma_C = 1$, "LCLCL").

5.7 Summary

In this chapter we introduce techniques for finding the worst-case makespan for a group of GPU threads: one approach which is pessimistic but has very low computational complexity and another approach (which builds on the former one) which employs Integer Linear Programming for an exact derivation (subject to some simplifying assumptions). Since the exact approach is computationally intractable for a large number of warps, we also introduce a simple way of obtaining, at only a fraction of the time, a safe estimate that is only marginally pessimistic.

6 Metaheuristic-based approach

We believe that instead of using computationally expensive techniques for finding an exact worst-case makespan, many soft real-time systems applications could benefit from a tight lower bound on the worst-case makespan. Hence, we would like to consinder estimation of the maximum makespan using *metaheuristics* – computational methods that try to find a better solution for an optimization problem iteratively, and statistically tend to converge to the global optimum over time. In the remainder of this chapter we present some considerations to motivate the idea behind the new technique. Sections 6.1 and 6.2 introduce the proposed metaheuristic. Sections 6.3 and 6.4 discuss the generation of suitable initial solutions and aspects of an efficient implementation, respectively. Section 6.5 provides a case study and some evaluation. Section 6.6 concludes.

6.1 Warp pseudo-precedence string

For the exact technique of the optimization-based approach in Chapter 5 (or in [22]), the objective is to maximize the makespan and the solution of an optimization problem is presented in the form of decision variables. For an approach using metaheuristics the objective remains to find the maximum makespan as well, but a question to consider is how to most conveniently represent the solution. One option is to express the solution in the form of the corresponding schedule as depicted in Figure 14.

A schedule representation not only contains all necessary information, such as the kernel instruction string, warp number, configuration of the streaming multiprocessor, the makespan, but it is also intuitive and readily understandable by humans. Still, we should check how suitable this representation is in the context of a metaheuristic that searches through a large solution space moving iteratively from the current solution to the neighbour solution, both being relatively "close" to each other. Let us apply

the concept of the *neighbour solution*, which is the core of the metaheuristics, to a schedule. If we move some instruction of some warp to a different clock cycle in the schedule in Figure 14, we can consider the resulting schedule in Figure 15 as a neighbour solution to the original one.

However, we can notice that in our example in Figure 15, just by moving that single instruction we are breaking the work-conserving property of the scheduling policy (at clock cycle 5 there is spare capacity of load/store units and a pending "L"-instruction for warps with the identifiers 1, 2 and 3, but the streaming multiprocessor is staying idle). This in turn makes the new solution invalid. The verification (regarding the precedence constraints or the work-conserving properties) of the altered schedule would be computationally expensive and there is no straightforward way of generating a priori valid schedules by moving instructions, other than validating *a posteriori*.

Clock Cycle	1	2	3	4	5	6	7	8
Warp 1	L	C			L			
Warp 2		L	C			L		
Warp 3			L	C			L	
Warp 4				L	C			L

Figure 14: Possible schedule ($\sigma_L = \sigma_C = 1$) as a valid solution

Clock Cycle	1	2	3	4	5	6	7	8	9
Warp 1	L	C							L
Warp 2		L	C			L			
Warp 3			L	C			L		
Warp 4				L	C			L	

Figure 15: An invalid solution (the work-conserving property is violated)

Therefore the schedule itself is probably not the best way of representing a solution, when using metaheuristics. For these purposes we therefore invented another data structure: the *warp pseudo-precedence string*. One possible way to derive the warp pseudo-precedence string from a schedule is the following: traversing the cells

of the schedule, column by column, from top to bottom, we append to an (initially null) integer string the identifier of the warp that performs some instruction in the corresponding clock cycle. For our example in Figure 14 the warp pseudo-precedence string is the following:

$$1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad 4 \quad 1 \quad 4 \quad 2 \quad 3 \quad 4 \tag{85}$$

Let us consider the warp pseudo-precedence string as a solution for the metaheuristics. To build a schedule from a warp pseudo-precedence string we simply traverse warp identifiers in the string one by one from left to right, and insert the corresponding instruction by the respective warp in the earliest clock cycle (i.e., in the left-most position in the schedule) possible, subject to capacity and precedence constraints. To determine whether this instruction is for a load/store unit or for a CUDA core, we need to keep track of how many instructions by each warp we have already scheduled at any instant. In other words, if we have already scheduled k instructions by the warp in consideration, then we need to examine the $(k+1)^{th}$ instruction of kernel instruction string, to see which computation unit should process it (e.g., if that is "L", or "C" etc.) The simple algorithm is presented in Figure 16.

```
//Warp pseudo-precedence string.
INPUT: warpPrecStr;
OUTPUT: schedule;

while (warpPrecStr is not fully traversed)
    //From warpPrecStr:
    w = read_current_warp_id();

    //According to the kernel instruction string:
    i = read_current_instruction_type_by_warp(w);

    //Subject to capacity and precedence constraints:
    t = find_earliest_cycle_wherein_possible_execute(w, i);

    add_to_schedule(w, i, t);
```

Figure 16: The algorithm for constructing the schedule.

We can try to get a neighbour solution by swapping the positions of warp identifiers in the string (85). There are many possible ways to do that, but let us consider moving all the identifiers of the warp 4 to the end of the string. After doing that, the warp pseudo-precedence string becomes "1 1 2 2 3 3 1 2 3 4 4 4". The schedule that corresponds to this new string as a neighbour solution is presented in Figure 17, and the makespan increases to 9 clock cycles.

Clock Cycle	1	2	3	4	5	6	7	8	9
Warp 1	L	C		L					
Warp 2		L	C		L				
Warp 3			L	C		L			
Warp 4							L	C	L

Figure 17: A valid neighbour solution (with increased makespan)

Hence, we can address the problem of estimating the maximum makespan from the following standpoint: find a warp pseudo-precedence string such that the corresponding makespan is maximized, subject to the configurations of the streaming multiprocessor under consideration.

One may notice that the warp pseudo-precedence string is a much more low-level representation (compared with the corresponding schedule), but because of the fact that it does not bind the warps to particular clock cycles, we are free to make permutations of the warps in the string subject to all the logic of the kernel, capacity, precedence and work-conserving constraints, even though these are not explicitly specified in terms of the data structure itself.

Because the warp pseudo-precedence string contains I instances (the number of instructions in the kernel instruction string) of each warp identifier (an integer in the range $[1, W]$), its length is $W \cdot I$ warp identifiers. In accordance with the multinomial theorem, the number of permutations of warp identifiers in the warp pseudo-precedence string is equal to $\frac{(W \cdot I)!}{(I!)^W}$. However, because of the fact that all the warps are identical, the equation above includes permutations that differ from each other only in terms of indexing of warps. For example if we swap the indexes of the warp 2 and the warp 3 in the permutation (85), we will get the following new permutation: 1 1 3 3 2 2 4 1 4 3 2 4, but the corresponding solution (as we consider it) is still the same. Every "unique permutation" corresponds to $W!$ permutations that could be obtained by re-indexing the warps. Hence the (still enormous) number of unique permutations is: $\frac{(W \cdot I)!}{(I!)^W \cdot W!}$ Note that even different unique permutations do not necessarily specify distinct solutions. For example, the warp pseudo-precedence string "1 2 1 3 2 3 1 2 3 4 4 4" still corresponds to the schedule in Figure 17 (built according to the different string "1 1 2 2 3 3 1 2 3 4 4 4").

6.2 The metaheuristic

As shown in Section 6.1, we present finding the maximum makespan as a combinatorial optimization problem where a solution is sought over a discrete search-space of warp pseudo-precedence strings. Considering even a relatively moderate length for the kernel instruction string (I), the brute-force search over $\frac{(W \cdot I)!}{(I!)^W \cdot W!}$ permutations would not be computationally tractable. Consequently, we apply computational methods that iteratively search for a "better" solution according to a given strategy. Among many different metaheuristics that are widely used in various scientific and application domains we decided in favour of simulated annealing by Kirkpatrick et al. [109], which is very popular for tackling combinatorial problems. Inspired by the annealing technique in metallurgy, simulated annealing attempts to replace the current solution of the problem with another candidate solution (often randomly obtained) at each its iteration. A candidate solution that improves on the current one is always accepted. However, occasionally, the algorithm will also accept a "worse" candidate solution with a probability which depends on the value of probability function. This function takes as parameters a variable T (also called as "the temperature") and the difference of the utilities of the current solution and the candidate solution. Higher temperatures and lower reduction in utility makes it likelier that such a candidate solution will be chosen. Occasionally accepting "worse" solutions helps avoid the pitfall of getting stuck at a local optimum of the optimization problem. With the number of iterations, T is decreased according to a given "annealing schedule".

Let $iter_{max}$ denote the (user-defined) maximum number of iterations for the annealing and let the variable $iter$ hold the index of the current iteration. Before the first iteration the temperature T is set to T_0 and is decreased after every iteration according to the following annealing schedule:

$$T = T_0 \cdot \left(1 - \frac{iter}{iter_{max}}\right)$$

The lower the temperature is set, the more "greedy" (in its preference for better solutions) the metaheuristic becomes. This principle is specified in the definition of the probability function which, besides T, also depends on the makespans of the current (m) and the candidate solution ($m^{cand.}$):

$$P(m, m^{cand.}, T) = \begin{cases} 1 & \text{if } m^{cand.} \geq m; \\ \min(1, \frac{T}{m - m^{cand.}}) & \text{otherwise.} \end{cases} \tag{86}$$

Note how the probability of accepting a solution with a smaller makespan decreases as $(m - m^{cand.})$ increases.

6.3 Providing a suitable initial solution

Although any "randomly" shuffled string consisting of I instances of each warp identifier could serve as an initial solution, providing a "good" initial solution to the metaheuristic may considerably speed up the convergence towards a good estimate of the makespan. Hence, although our technique is parallelizable over an arbitrary degree of processors (which would help with convergence speed), we present some "templates" (according to our empirical observation) for generating initial solutions with long makespan. When running the metaheuristic on a multi-processor machine (with one thread per processor), we recommend using the initial solutions presented below on some processors and random warp pseudo-precedence strings on the rest.

6.3.1 "Round-robin"

The corresponding warp pseudo-precedence string can be constructed based on the following pattern:

$$\underbrace{\underbrace{1, 2, \ldots, W,}\underbrace{1, 2, \ldots, W,} \ldots, \underbrace{1, 2, \ldots, W}}_{\text{I times}}$$

An example of a schedule generated using a "round-robin" pseudo-precedence string is the one in Figure 14.

6.3.2 "Fixed-priority"

Clock Cycle	1	2	3	4	5	6	7	8	
Warp 1	L	C	C	L					
Warp 2		L		C	C	L			
Warp 3			L			C	C	L	

Figure 18: Fixed-priority ($\sigma_L = \sigma_C = 1$)

The respective warp pseudo-precedence string could be easily constructed according to the pattern presented below:

$$\underbrace{1, 1, \ldots 1}_{\text{I times}}, \underbrace{2, 2, \ldots 2}_{\text{I times}}, \ldots \underbrace{W, W, \ldots W}_{\text{I times}}$$

Using a "fixed-priority" pseudo-precedence string outputs the schedule that we would get if warps were assigned static priorities and dispatched under those (as in Figure 18).

6.3.3 Most Pending Warp Executes First

Clock Cycle	1	2	3	4	5	6	7	8
Warp 1	L	C		C	L			
Warp 2		L	C			C	L	
Warp 3			L		C		C	L

Figure 19: Most Pending Warp Executes First ($\sigma_L = \sigma_C = 1$)

To construct such a schedule (and, eventually, a corresponding warp pseudo-precedence string), we need to maintain a list of "pending" warp identifiers, initialized as $\langle 1, \ldots W \rangle$. The schedule for clock cycle t is constructed before moving on to cycle $t+1$. To schedule a warp within a given clock cycle, the algorithm traverses the list from head to tail (i.e., left to right) until it finds a warp which could be scheduled in that given cycle, subject to the availability of free processing units. As soon as that instruction is inserted into the schedule, the index of the corresponding warp is appended to the (initially empty) warp pseudo-precedence string and the same warp index is removed from its position in the list and inserted at the tail of the list. If all processing units are made busy for the current clock cycle or when all element of the list have been traversed, the algorithm moves on to the next clock cycle. This algorithm is presented in pseudocode in Figure 20.

```
INPUT: kernInstrStr; //Kernel instruction string.
OUTPUT: warpPrecStr; //Warp pseudo-precedence string.

//List of identifiers of pending warps.
pendWarpList = < 1, 2, ...W >;
clockCycle = 1; //The first clock cycle.
while (pendWarpList is not empty)
    index = 1; //The first index in pendWarpList.
    while (exists spare capacity and unread warps)
        w = read_warp_id_at(index); //From pendWarpList
        //According to kernInstrStr
        i = read_next_instruction_type_by_warp(w);
        //Subject to capacity constraints
        if (exists spare capacity of i at clockCycle)
            warpPrecStr += w;
            remove_warp_id(w); //From pendWarpList
            if (warp w does not finish execution)
                insert_warp_id(w); //To pendWarpList
    clockCycle += 1;
```

Figure 20: Constructing a "Most Pending Warp Executes First" initial solution.

As an illustration, consider the example in Figure 19: the list is initially ⟨1,2,3⟩. By scheduling warp 1 in clock cycle 1, it becomes ⟨2,3,1⟩. But warp 2 cannot be scheduled within the same cycle due to capacity constraints; not can warp 3. Therefore, we move to clock cycle 2 (the list is still ⟨2,3,1⟩). We can schedule warp 2 in this cycle and the list becomes ⟨3,1,2⟩. Then, warp 3 is not schedulable in cycle 2, but warp 1 is (hence, the list becomes ⟨3,2,1⟩. And so on.

6.4 Implementation optimization

For each new candidate solution considered, the metaheuristic needs to create a corresponding schedule from the new warp pseudo-precedence string under consideration using the algorithm of Figure 16, so that the corresponding makespan can be calculated. Doing so from scratch could be an option, but would be inefficient, in the sense that, if each neighbour solution was obtained just by a single permutation (or a few) of the warp pseudo-precedence string, then surely the two schedules would be similar and, in principle, there should exist a faster way, of deriving the one from the other by doing just the part of the computation reflecting the differences of the two pseudo-precedence strings. Over a large number of iterations the time saved would be significant (and the convergence to a good estimate of the makespan would be sped up). Therefore, we introduce the warp cycle string $warpCycleStr$ – an integer string of the same length as the warp pseudo-precedence string $warpPrecStr$. Element $warpCycleStr[w]$ holds the index of the clock cycle in which the warp with the identifier $warpPrecStr[w]$ is scheduled. The $warpCycleStr$ itself is a "compact" way of storing a schedule (instead, e.g., of sparse two-dimensional arrays). If the first index where the new $warpPrecStr$ differs from the previous one is z, then, from elements $warpCycleStr[1]$ to $warpCycleStr[z-1]$ we can obtain the "common" part of the schedule. It then suffices to assign new values for elements $warpCycleStr[z]$ onwards, considering the rest of the new pseudo-precedence string (i.e., from $warpPrecStr[z]$ onwards).

6.5 Case studies

The technique presented in Section 5.5 can be used for finding in tractable time an upper bound on the worst-case makespan. The metaheuristic-based approach can supplement that technique, in an analysis tool, for also bounding the worst-case makespan from below. We implemented this as multithreaded module.

6.5.1 Overview

Parameters for the problem instance under consideration can be categorized as (i) program-related (the number of warps; the kernel instruction string), (ii) hardware-related (the number of computational units of each type; the warp size) and (iii) metaheuristic-related (the initial temperature T_0, the maximum number of iterations $iter_{max}$ and an integer flag specifying the kind of the initial solution – i.e., whether it is random or obtained according to one of the patterns presented in Section 6.3). These parameters serve as input to each thread (on the respective processor), which then starts to iterate among candidate solutions, in parallel with (and independently of) other threads on other processors. The estimate, at any instant, is obtained as the greatest reported makespan so far, over all threads.

6.5.2 The benchmark

For our experiments, we choose a kernel instruction string derived from a real application that could be run as many parallel GPU threads: Voronoi diagrams [179] which are used e.g., for solving proximity problems in computational geometry or localization in wireless sensor networks.

A Voronoi diagram on a two-dimensional plane, like the one depicted in Figure 21, consists of polygonal tiles, each "centered" around a corresponding *limit point*. Each tile consists of the points in the plane closer to the particular limit point than to any other. Segments in a Voronoi diagram are formed from the points of the plane which are equidistant to two neighboring limit points.

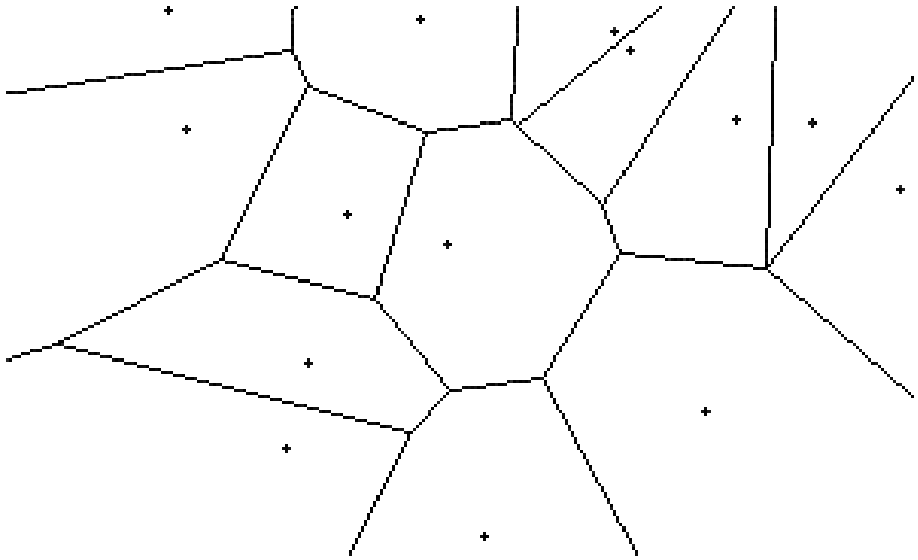


Figure 21: Voronoi diagram for a set S of limit points.

For practical implementations (such as visualization on a screen), the concept can be extended from a plane to rasters with a finite number of points (pixels). For those cases, the algorithm in [141] is much easier to implement than the one presented by Shamos and Hoey [171] (and based on a divide-and-conquer paradigm) or Fortune's sweepline algorithm [65].

For every pixel{*Calculate distance to every limit point;*
Select the closest limit point;
Put the pixel into conformity with that limit point;}

The iterations for each pixel are entirely independent, permitting a high degree of parallelism. In C-like pseudocode, one iteration may be presented as in Figure 22, with each thread given the coordinates (x, y) of a pixel and computing the distance to every limit point (x_i, y_i) in a set S.

```
//minimal distance square
float md = (x − x_1)^2 + (y − y_1)^2;
//minimal distance point
int mdp = 1;
// N is number of points in S;
for (int i=2; i<=N; i++)
  if ( (x − x_i)^2 + (y − y_i)^2 < md ) {
    md = (x − x_i)^2 + (y − y_i)^2 ;
    mdp = i;}
```

Figure 22: Simple Voronoi diagram representing code.

Our "port" of that program to assembly for NVIDIA's Parallel Thread Execution (PTX) virtual machine [158] is shown in Figure 23. Every line consists of an assembly statement, comments that "map" that statement to the corresponding code from the original higher-level program illustrated in Figure 22 and a character for the type of hardware unit assumed to perform the corresponding assembly instruction. We tag instructions executed on CUDA core with a "C" and instructions for a load/store unit with an "L". The resulting kernel instruction string corresponding to the branchless code, from the start of the program until the end of the first iteration of the inner loop in Figure 23, was used in our experiments.

```
        mov.u32     $r0,   N_addr;          // N                                  L
        mov.f32     $f1,   x_addr;          // x                                  L
        mov.f32     $f2,   y_addr;          // y                                  L
        mov.f32     $f3,   x1_addr          // x1                                 L
        mov.f32     $f4,   y1_addr;         // y1                                 L
        sub.f32     $f5,   $f1,   $f3;      // (x-x1)                             C
        mul.f32     $f6,   $f5,   $f5;      // (x-x1)^2                           C
        sub.f32     $f7,   $f2,   $f4;      // (y-y1)                             C
        fma.f32     $f8,   $f7,   $f7,  $f6; // (x-x1)^2+(y-y1)^2                 C
        mov.f32     $f0,   $f8;             // md = (x-x1)^2+(y-y1)^2             C
        mov.u32     $r1,   1;               // int mdp = 1;                       C
        mov.u32     $r2,   2;               // for (int i=2; i<=N; i++)           C
Loop:   setp.gt.u32 p,     $r2,   $r0;      // for (int i=2; i<=N; i++)           C
@p      bra Done;                           // for (int i=2; i<=N; i++)           C
        mov.f32     $f3,   xi_addr;         // xi                                 L
        mov.f32     $f4,   yi_addr;         // yi                                 L
        sub.f32     $f5,   $f1,   $f3;      // (x-xi)                             C
        mul.f32     $f6,   $f5,   $f5;      // (x-xi)^2                           C
        sub.f32     $f7,   $f2,   $f4;      // (y-yi)                             C
        fma.f32     $f8,   $f7,   $f7,  $f6; // (x-xi)^2+(y-yi)^2                 C
        setp.ge.f32 q,     $f8,   $f0;      // if ( (x-xi)^2+(y-yi)^2 < md)       C
@q      bra If;                             //                                    C
        mov.f32     $f0,   $f8;             // md = (x-xi)^2+(y-yi)^2 ;           C
        mov.u32     $r1,   $r2;             // mdp = i;                           C
If:                                         // if ( (x-xi)^2+(y-yi)^2 < md)
        add.u32     $r2,   $r2,   1;        // for (int i=2; i<=N; i++)           C
        bra Loop;                           //                                    C
Done:
```

Figure 23: PTX program for visualizing Voronoi diagrams.

6.5.3 Experimental results

The metaheuristic approach described outputs a lower bound on the worst-case makespan for the problem instance in consideration under the simplifying assumptions discussed earlier. These assumptions were all pessimistic except for the assumption that all load/stores are single-cycle. Conversely, the optimization-based approach outputs an upper bound for the worst-case makespan under the same assumptions. Therefore, we sought to investigate the "quality" of the solutions output by the metaheuristic by comparing its output with that of the optimization-based approach in Chapter 5.

As a benchmark, we used the Voronoi kernel instruction string introduced earlier:

$$\underbrace{LLLLL}_{5\ Ls}\underbrace{CCCCCCCCC}_{9\ Cs}\underbrace{LL}_{2\ Ls}\underbrace{CCCCCCCCC}_{9\ Cs}$$

We used parameters $\{\sigma_C = 4,\ \sigma_L = 1\}$ (intended to model NVIDIA Kepler, under the pessimistic assumption that only one instruction dispatch unit per warp scheduler is used) and for $W = 16$ warps. We ran 8 instances (Java threads) of the metaheuristic (2 with the "round-robin" initial solution; 2 with "fixed-priority"; 2 with "most pending warp executes first"; 2 random) with initial temperature $T_0 = 0.3$ for $2 \cdot 10^6$ iterations each on a Pentium Dual-core E5400 (2.7 GHz). These runs were performed sequentially, not in parallel. However, by logging every reported improvement to the current estimate along with timestamps, in seconds since the beginning, we were able to retroactively "simulate" the behavior one would get by running the instances of the metaheuristic in parallel, since their executions would be independent anyway. The reported estimates of the individual Java threads are plotted in Figure 24, with the horizontal axis denoting the time since launch. The composite reported estimate, obtained as the maximum over all graphs, at any time instant (i.e., as the "envelope" of all graphs), converged to 160 at the end of the experiment.

By comparison, the upper bound on the worst-case makespan obtained via the optimization-based approach for 16 warps was 176 clock cycles and took 58 hours to compute, on the same machine. It was derived by pessimistically extrapolating from the respective exact worst-case estimate for 4 warps, which was the most that could tractably be computed. This means that the estimate by the metaheuristic was just 9.1% lower than the one by the optimization-based approach. We interpret this as evidence that the both approaches provide relatively tight lower/upper bounds respectively for the worst-case makespan, subject to our assumptions. However, the metaheuristic provides its estimates orders of magnitude faster.

Additional observations from this small-scale experiment are that, even the "round-robin" initial solution can serve as a quick/rough estimate for the worst-case makespan (even *before* running the meta-heuristic). This is also in accordance with our experience by experimenting with other kernel instructions strings and problem instances in general. However, even when the the metaheuristic is launched with random initial solutions, it converges fast towards better estimates, comparable to those obtained when using the "round-robin" initial solution. The graphs also serve, to an extent, to highlight the relative speedup that can be achieved in the convergence to a good estimate, by running (and tracking) multiple independent instances of the metaheuristic in parallel.

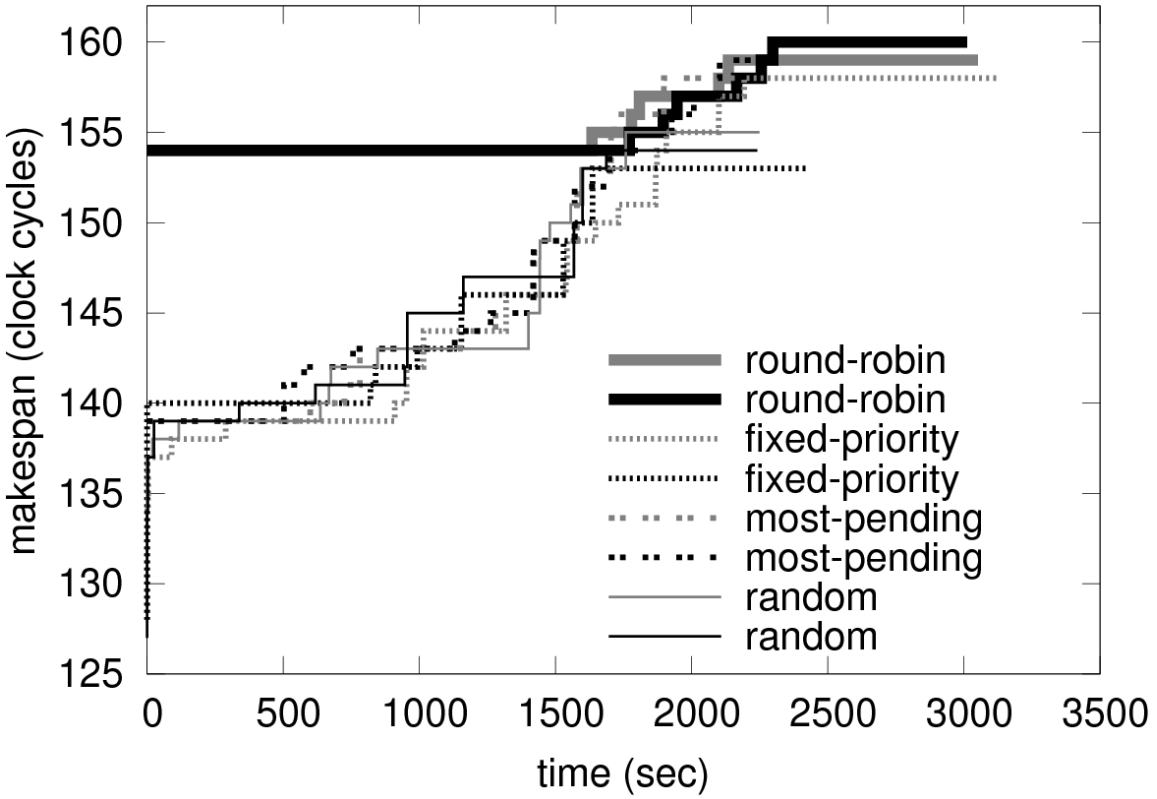


Figure 24: Convergence of the estimates of the worst-case makespan over time, for 8 instances of the metaheuristic, with different initial solutions.

6.6 Summary

This chapter presents an approach for tractably obtaining an estimate of the worst-case makespan of a set of identical GPU threads running on a single streaming multiprocessor, subject to some simplifying assumptions. This approach is based on the metaheuristic of simulated annealing and is readily parallelizable, for even faster convergence. The result is very close to the pessimistic estimate obtainable using much more computationally complex optimization-based approach presented in Chapter 5. Therefore, the estimate output by this metaheuristic-based approach is, in the most unfavourable circumstance, a slight underestimation of the actual worst case. As such, the target of the approach is soft-real time systems, wherein a very rare missed deadline does not matter.

7 Statistical measurement-based approach

Our prior approaches presented in Chapter 5 and in Chapter 6 for deriving WCET estimates for GPU kernels were optimistic in their assumptions on cache misses and the memory subsystem in general. Extending them, as originally intended, to also consider the effects of cache and memory, turned to be challenging for two reasons. First, due to tractability issues, inherent in those approaches, which kicked in when considering long-latency operations (e.g., hundreds of cycles for an L1 miss). Secondly, because the exact cache architectures and replacement policies for modern GPUs are trade secrets, thus not openly documented. A probabilistic measurement-based approach bypasses both hurdles. To that end, we undertake a measurement-based probabilistic approach, based on Statistic Analysis and Extreme Value Theory (EVT). This technique allows the derivation of highly accurate estimates on the probability that any run of the GPU application exceeds a respective time threshold, even if such high execution times are not observed in any of the measurements. It advances the state of the art because it accurately captures the overall behaviour of the memory subsystem. In terms of outline, Section 7.1 elaborates on measurement collection. Section 7.2 offers background on the statistical analysis of the measurements and on EVT, which we use to obtain highly accurate probabilistic WCET estimates. Section 7.3 discusses our experiments. Section 7.4 concludes.

7.1 On Collecting Measurements

In the typical CUDA setup, the following sequence of actions is performed by a CUDA-C program [101].

S1: The program allocates memory on the host (CPU) for the input and output of the CUDA kernel.

S2: The program allocates[7] memory on the GPU (device) for the input and output of the CUDA kernel.

S3: The program initiates[7] the copying of the input from host memory to GPU memory. This is normally a blocking operation, unless the copied data is less than 64KB [155].

S4: The program launches the CUDA kernel. This operation is non-blocking: the driver returns control to the CPU immediately after the launch[8].

S5: The kernel executes on the GPU until completion. In parallel, the program on the host polls on the status of its completion.

S6: Upon completion of the kernel, the program copies[7] the output of the CUDA kernel from GPU memory to host memory.

S7: The program continues its execution on the host.

The execution time of the kernel corresponds to stage S5. Let that be denoted as T^{DEV}. However, the combined duration of stages S2 to S6 is also of interest, since it determines the acceleration attained via CUDA. Let us denote that by T^{HOST}. If determining T^{DEV} analytically (which our approaches in Chapter 5 and Chapter 6 attempted) is hard, for T^{HOST} it is even more so, since it also includes the execution of the CUDA driver and the I/O latency for copy over the PCI-e bus. Therefore, we attempt to characterise both by collecting measurements over a sufficiently large number of runs and applying EVT.

To measure T^{HOST} we used standard Linux primitives for reading the system time. We placed those system calls just before S2 and at the start of S7. Accurately measuring T^{DEV} is harder because the GPU cannot be probed. Any instrumentation

[7]Via the high-level CUDA Run-time API or directly the Driver.

[8]Synchronous semantics (i.e., self-suspension until the GPU-side computation completes) can still be obtained, e.g., via custom CPU-side programming.

code added to the kernel would be executed by *all* CUDA threads so it would have to be extremely light-weight/non-intrusive for the cumulative effect on T^{DEV} to not be significant. Recall that T^{DEV} is the interval *from* when the first kernel instruction by some thread (warp) executes *until* the last kernel instruction by some thread (warp) is completed. The tricky part is that we cannot know *a priori* which warp starts to execute first and which one completes last. We deal with this as follows:

There is a special *clock-register* on each SM, which counts GPU cycles. We read/record its value via manually inserted assembly code, at the start/end of each thread. A *naive* approach would use two respective *per-thread* variables, start_cycle and end_cycle. But this would use too much shared memory (out of the 48 KBs, at most, per SM) or else thrash the L1 cache, significantly altering the timing behaviour. Hence we use a single *per-SM* pair of start_cycle and end_cycle variables (Figure 25), and leverage the fact that execution on the GPU is in-order. The first thread to execute, whichever that is, sets the start_cycle variable. All subsequent threads detect this (if-condition at line 1) and avoid overwriting its value. Upon completion, all threads write to the stop_cycle variable (line 4), which means that the last value written to it is by the latest thread to complete. Then T^{DEV} (in GPU cycles) is derived[9], with p denoting the index of the SM, as:

$$T^{\text{DEV}} = \max_{p}\{\text{end\_cycle}[p]\} - \min_{p}\{\text{start\_cycle}[p]\} \tag{87}$$

To apply EVT, we need such measurements from many runs of a given CUDA kernel. We therefore developed a tool that repeatedly (i) launches the same kernel and (ii) records its timing measurements. To eliminate interference from screen rendering, we switch off the windowing system entirely. To guarantee the safe application of the EVT, the number of runs must be large enough; in the order of thousands, as has

[9]In the rare case of clock-register overflow, the above code does not work. We detect/discard such data, offline.

```
    //start_cycle initialised to MAXINT
    //if-condition TRUE only for the earliest thread
1.  if (start_cycle>CLK_REG)
2.   start_cycle:=CLK_REG;
3.   —– (The instructions of the kernel go here...) —-
4.  stop_cycle:=CLK_REG; //overwritten by every
    thread
```

Figure 25: High-level overview of the measurement-collecting assembly inserted in each GPU thread.

been demonstrated. We conservatively opted for 10^5 runs, which, as expected, proved to be more than enough.

7.2 Statistical Analyses of Execution Time

Statistical estimations of worst-case execution time are becoming popular within the real-time community, [53, 26, 83, 37, 48]. They lead to the notion of probabilistic WCET (pWCET), alternative to the deterministic WCET, as distributions of values C_j with an associated probability of being the WCET. C_j upper-bounds the task execution time with a probability p_j. $1 - p_j$ is the probability for a task instance having a bound on its execution time different than C_j.

Definition 1 (probabilistic WCET)**.** *Given $\mathcal{C}_i$, the distribution of execution time measured in a certain configuration/condition i, the probabilistic Worst-Case Execution Time distribution $\mathcal{C}^*$ of a task is a tight upper bound on the execution time distribution $\mathcal{C}_i$ of all possible execution conditions*[10]*. Hence, $\forall i$, $\mathcal{C}^*$ is larger than or equal to $\mathcal{C}_i$. In notation: $\mathcal{C}^* \succeq \mathcal{C}_i \ \forall i$.*

The total ordering among distributions is defined such that, a distribution $\mathcal{C}_j$ is greater than or equal to a distribution $\mathcal{C}_k$, $\mathcal{C}_j \succeq \mathcal{C}_k$, iff $P\{\mathcal{C}_j \leq d\} \leq P\{\mathcal{C}_k \leq d\}$ for any d and the two random variables are not identically distributed (two different

[10]We use calligraphic letters to represent probability distributions. Non calligraphic letters are for single values.

distributions), [52]. The tightest possible pWCET distribution would be the exact pWCET, which is unknown. However, we still need to come up with a *safe* pWCET estimation, meaning a pWCET estimation $\mathcal{C}^*$ that is greater than or equal to the (unknown) exact pWCET. And the only information we can rely on, for constructing such a pWCET estimation is the set of measurements ($\{\mathcal{C}_i\}$) and the execution conditions (i) under which they were taken.

The probabilistic worst-case execution time can also be defined in terms of the exceeding thresholds and the 1-Cumulative Distribution Function (1-CDF) representation. Given a probability of exceedence p^*, C^* is the worst-case execution time such that $P\{\mathcal{C}^* \geq C^*\} \leq p^*$. Alternative to the pWCET distribution, we can call minimum probabilistic worst-case execution time the tuple $\langle C^*, p^* \rangle$. In our experiments we consider $p^* = 10^{-6}$, $p^* = 10^{-9}$, and $p^* = 10^{-12}$.

Measurements, when used in conjunction with statistical approaches such as the EVT, contribute at estimating safe pWCETs. On their own, measurements are *not* enough to obtain pWCETs since they may lack completeness: through the measurements there is no guarantee to have experienced all the execution conditions. Nonetheless, measurements are important for extracting observable features such as average behaviours and trends that can appear while executing tasks. Extreme value analysis is for the statistical inference on the tail region of a distribution function. The statistical estimation of the pWCET makes use of the EVT for exploring rare events, wherein the WCET and its probabilistic version pWCET should lie. In the following we state the basics for the EVT that we apply in our framework.

Classical EVT discusses the possible limiting laws for the maximum

$$M_n = max\{X_1, X_2, \ldots, X_n\}$$

of n independent identically distributed (i.i.d.)[11] random variables $\{X_n\}$ as n tends

[11]Readers not already familiar with the concept of independent and identically distributed vari-

to infinite[12]. [80].

Theorem 1 (Fisher-Tippett-Gnedenko EVT)**.** *Let* $X_1, X_2, \ldots, X_n$ *be a sequence of independent and identically-distributed random variables, and* $M_n = \max\{X_1, \ldots, X_n\}$*. If a sequence of pairs of real numbers* a_n, b_n *exists such that each* $a_n > 0$ *and*

$$\lim_{n\to\infty} P\left\{\frac{M_n - b_n}{a_n} \leq x\right\} = \mathcal{G}(x), \tag{88}$$

where $\mathcal{G}$ *is a non degenerate distribution function, then the limit distribution* $\mathcal{G}$ *belongs to either the Gumbel, the Frechet or the Weibull family. These can be grouped into the generalised extreme value distribution.*

Theorem 1 expresses the EVT theory in case of independence among samples: the maxima of an i.i.d. sequence converge to a Generalised Extreme Value (GEV) distribution $\mathcal{G}_\xi$, which admits the following Cumulative Distribution Function (CDF):

$$\mathcal{G}_\xi(x) = \begin{cases} \exp(-\exp(-x)), & \text{if} \quad \xi = 0 \\ \\ \exp\left(-(1+\xi x)^{-\frac{1}{\xi}}\right), & \text{if} \quad \xi \neq 0 \end{cases}. \tag{89}$$

The GEV distribution $\mathcal{G}_\xi$ can be of three distinct types, characterised by $\xi = 0$, $\xi > 0$ and $\xi < 0$, which correspond to the Gumbel, Fréchet and Weibull distributions, respectively.

Usually, the EVT is established for i.i.d. observations, and previous works have linked the safety of EVT estimations to that hypothesis. Therein, it is claimed that if both independence and identical distribution are verified, the EVT distribution tail

ables, may peek ahead to Sections 7.2.1.1 and 7.2.1.2, where we formally define and discuss these concepts.

[12] $\{X_n\}$ is the sequence of observations; each observation results from a distribution $\mathcal{X}_n$. The identical distribution hypothesis assumes that all the observations follow the same distribution, thus $\mathcal{X}_1 = \mathcal{X}_2 = \ldots \mathcal{X}_n = \mathcal{F}$. In our case, both observations and distributions refer to execution time, hence there is equivalence between $\{X_n\}$ and $\{C_n\}$ as well as $\mathcal{X}_n$ and $\mathcal{C}$, in terms of representation.

projection can be considered as a safe pWCET estimation, [48].

However, more recent developments showed that independence is *not* a necessary hypothesis for the EVT. Leadbetter et al. [114], Hsing [92] and Northrop [151] developed EVT for stationary weakly dependent time series. The latter two references also established statistical tools for use under that assumption.

Theorem 2 (Long Range Independence EVT, [115])**.** *Let* $\{X_n\}$ *be a stationary sequence such that*
$M_n = max\{X_1, \dots X_n\}$ *has a non-degenerate limiting distribution* $\mathcal{G}$ *as in*

$$P\{a_n(M_n - b_n) \leq x\} \xrightarrow{d} \mathcal{G}(x), \tag{90}$$

for some constants $a_n > 0$, b_n*. Suppose that*

$$D(u_n): \ |F_{i_1,\dots,i_p,j_1,\dots,j_q}(u_n) - F_{i_1,\dots,i_p}(u_n) \cdot F_{j_1,\dots,j_q}(u_n)| \leq \alpha_{n,l},$$

where $lim_{l\to\infty} lim_{n\to\infty} \alpha_{n,l} = 0$*, holds for all sequences* u_n *given by* $u_n = x/a_n + b_n$*,* $-\infty < x < \infty$*. Then* $\mathcal{G}$ *is one of the three classical types: Weibull, Frechet, Gumbel.*

The distributional mixing condition $D(u_n)$ alone is sufficient to guarantee that the central classical result concerning the possible extremal types (the EVT), holds also for stationary sequences. Both a_n and b_n can be computed as best-fit of the input observations. $D(u_n)$ is called long-range dependence conditions, and if satisfied it means that there is no dependence between far away observations.

In [115] it is introduced the local dependence condition $D'(u_n)$,

$$D'(u_n): \ lim_{n\to\infty} \sup n \cdot \sum_{j=2}^{n/k} P\{X_1 > u_n, X_j > u_n\} \to 0$$

slightly more constraining than $D(u_n)$, seeking to assure the independence between close-in-time observations. If $D'(u_n)$ holds with $k \to \infty$ and for each $u_n = x/a_n + b_n$,

then the particular distribution type which applies is the same as if the sequence $\{X_n\}$ were i.i.d, with the same marginal distribution function, and the same normalizing constants a_n, b_n may be used.

Theorem 3 (Extremal Independence EVT, [115]). *Let $\{X_n\}$ be a stationary sequence with marginal distribution function $\mathcal{F}$ such that $M_n = max\{X_1, \dots X_n\}$, and $\{u_n\}$ a sequence of constants such that $D(u_n)$, $D'(u_n)$ hold. Let $0 \leq \tau < \infty$, then*

$$P\{M_n \leq u_n\} \xrightarrow{d} exp(-\tau) \tag{91}$$

iff

$$n \cdot [1 - F(u_n)] \to \tau. \tag{92}$$

Theorem 3 states that if both $D(u_n)$ and $D'(u_n)$ are satisfied, the resulting EVT is equal to the one obtained in case of observing independence.

Chernick [41], extending Loynes [137], showed that, if for each $\tau > 0$, $u_n = u_n(\tau)$ is defined to satisfy Equation (92), under $D(u_n)$ conditions alone, then any limit function for $P\{M_n \leq u_n(\tau)\}$ must be of the form

$$P\{M_n \leq u_n(\tau)\} \xrightarrow{d} exp(-\theta\tau), \tag{93}$$

for some θ with $0 \leq \theta \leq 1$.

The parameter θ, called the *extremal index* of the time series, is a measure of clustering at the extremes. It is useful for analysing the behaviour of the extremes in the tail; a small θ means greater clustering of the largest observations, i.e., higher dependence between observations; a value of $\theta = 1$ i.e., no extremal clustering, denotes independence.

Assuming $\overline{\mathcal{C}}$ the pWCET EVT estimation in case of stationarity, and $\widehat{\mathcal{C}}$ the

pWCET EVT estimation in case of independence. Supposing that the execution time measurements in the two cases follow the same marginal distribution, it is $\overline{\mathcal{C}} = \widehat{\mathcal{C}}^{\theta}$ with $\overline{\mathcal{C}} \succeq \widehat{\mathcal{C}}$, [38]. In case of independence at the extremes, $\theta = 1$, $\overline{\mathcal{C}} \approx \widehat{\mathcal{C}}$, [38]. Once one of the above hypotheses (either independence, extremal dependence, or stationarity) is satisfied the EVT provides pWCET estimations which are greater than or equal to the exact pWCET. In here, the safety of pWCET EVT estimations.

In the present chapter we apply these theoretical developments to the execution time analysis and safe pWCET estimations. In doing so, we consider the Gumbel distribution for EVT pWCETs, as it has been demonstrated to be the most appropriate distribution for execution times, [48].

7.2.1 On the Verification of the EVT hypotheses

Hypothesis testing means to decide, from a number of observations, whether one should consider a property to be true or not. We may never know for sure, but a statistical test will give us guidance in making a decision. In statistics we can state this problem using two hypotheses: H_0 (named null hypothesis) that denotes the hypothesis that the property is true, and H_1 (namely alternative hypothesis) denoting the hypothesis that the property is false. It has to be decided whether to accept or reject the hypothesis H_0 based on a sample (set of observations). The ρ-value is the result for hypothesis testing, where ρ is the probability of obtaining a test result at least as extreme as the one that was actually observed, assuming that the null hypothesis is true. Normally, $\rho > 0.05$ validates H_0; $\rho \leq 0.05$ rejects H_0, thus validates H_1. Various alternative approaches exist for calculating such an ρ-value, leading to different hypothesis tests; we discuss, later on in this section, those ones that we will be using.

7.2.1.1 Independence of Observations In statistics, a collection of random variables is independent (i.) if all the random variables are mutually independent. By this, we mean whether individual observations within the same execution trace are correlated with each other or not. If knowing one observation tells you something about another, then the observations are dependent; if knowing one observation tells you nothing about another, in that case they are independent.

A test applied in [48] aims at proving that samples are independent looking for randomness. This is called *runs test*, where randomness is sought within the observed data series by examining the frequency of "runs"; a "run" is a series of similar responses.

In this chapter we look to extend independence tests from runs test, since randomness is not formally sufficient to verify independence. This type of independence can not be proven or tested except for time series. Time series tests are based on autoregression and autocorrelation. In particular, we aim at verifying stationarity, which gives more information about the observation traces and applying it to characterise system execution behaviour while looking for the worst-case execution conditions.

7.2.1.2 Identical Distribution of Observations In statistics, a collection of random variables is identically distributed (i.d.) if each random variable has the same probability distribution. A common test for verifying identical distribution in observations is the two-sample *Kolmogorov-Smirnov* test: The trace of observations is divided into two sets which are compared, to verify whether they represent the same distribution.

7.2.2 Statistical Analyses

In practical applications, the independence assumption may or may not be realistic. To test how realistic it is on a given execution time data set, the autocorrelation can

be computed with *lag plots*, or a *turning point test* can be performed. These are to test the relationship that exists between measured observations. We apply them in order to extract patterns and behavioural models which could describe the observed system behaviour. In particular, we employ autocorrelation tests together with the notion of stationarity, which indirectly quantify the statistical independence between observations.

With no means of formalism, a process is stationary if its mean variance and autocovariance structure do not change over time. This is what is called weak form of stationarity, which means flat-looking observations, no trend, constant variance over time, and no periodic fluctuations or autocorrelation.

Autocorrelation, in a time series, is the similarity between observations as a function of the time lag between them. In our case, the time is given with the order of observations, thus lags are in terms of number of observations. The sample Auto Correlation Function (ACF) is one of the most important assessment tools for detecting data dependence and fitting models to data. Although the model is not faced at first, the observed data $\{X_1, \ldots, X_N\}$ are known.

An *autoregressive (AR) model* instead, is a representation of a type of random process. The AR model describes the underlying stationarity model of a trace of observations (time series): $AR(0)$, the sequence of observations has no dependence between the observations – white noise; $AR(1)$, a process where, with a positive parameter, only the previous observation in the process and the noise term contribute to the output – very light dependence; $AR(2)$, a process where the previous two observations and the noise term contribute to the output. And so it goes on, increasing the dependence pattern between observations.

We also use the *Ljung-Box* test, which looks for any significant evidence for non-zero correlations between lags. Large ρ-values from the test suggest that the series is not stationary, thus there is no trend between consecutive observations; this supports

non stationarity, and thus independence.

Valuable to time series analysis is also the test called *extremogram* [49], where the dependence at the extremes is estimated. The extremogram defines an analogue of the autocorrelation function, which depends only on the extreme values in the sequence of observations.

Finally, to compare with the independence case, there is the extremal index θ of the observations which is another tool for measuring the dependency of extreme values. We make use of the *blocks test* to compute θ, based on estimators [57]. We stress that there is equivalence between the extremogram and the extremal index, for evaluating extremal dependences, thus ultimately the EVT applicability. For completeness we apply both, although just one of the two would have been enough to verify extremal behaviour of observations.

Tests such as the above allow us to conclude about the stationarity of execution time observations and their eventual extremal dependence. As earlier argued, under those circumstances it *is* still possible to derive safe EVT distributions, thus safe pWCET estimations. More importantly, the stationarity helps with describing the execution behaviour and points out to us which are the worst-case conditions necessary, in order to safely conclude about pWCETs.

7.3 Experiments

Our testbed used a Kepler GK104 with 8 SMs (Figure 5), configured with 32KB of shared memory and 32KB of L1 each. As benchmark, we developed in CUDA a Voronoi diagram generator, according to the raster-coloring massively parallel approach [141] also used in Section 6.5.2 of the metaheuristic-based approach in Chapter 6. Informally, a Voronoi diagram for a 2D-plane and K points on it, divides the plane into tiles, each tile consisting of the points in the plane closer to one of the K points than to any other. For a 2D-raster, this is formed by calculating, for every

pixel, the distance to each of the K points. Our application uses a separate thread per pixel. Therefore, the raster size (X by Y) determines the number of threads, whereas the number of points K determines the workload of a thread. For valid comparisons (same per-thread workload) we used $K = 32$ in all setups and simply varied the number of threads. The first setup (VOR-1) used X=Y=32 which corresponds to 1024 threads (32 warps), the maximum thread block size in Kepler. The other setups involved 8, 28 and 32 thread blocks of this size. The execution times are in ns.

7.3.1 Timing Analysis

The experiments made provide execution time measurement traces, to be statistically tested. As we will proceed to show, although the T^{DEV} and T^{HOST} traces behaved very differently, all traces, upon testing, indeed support the conditions that permit the safe application of EVT.

Table 1 groups the numerical results of the independence tests carried out, i.e., runs test (runs), Ljung-Box (LB), and autoregressive (AR). These results reveal the **independence** of the T^{DEV} case; hence realistic cases could be independent, and the EVT could be applicable with no need for artificially induced randomicity, as made in [48] with random replacement caches. Instead, the T^{HOST} traces are **not independent, but stationary**. This is due to the filtering effects that HOST exercises, which reduce variability and thus the independence of the observations. This stationarity is present at different degrees in the 4 different traces of T^{HOST}, but EVT is still applicable to all of them (Equation (93)).

The combination of the autocorrelation tests, the stationary tests and the extremogram (Figure 28) gives more accuracy and completeness to the independence/ stationarity verification than just the runs test. For example, in case of VOR-32 T^{HOST}, the runs test would have concluded about the trace independence; in reality though, it exhibits stationarity – and, in particular, a strong stationary relationship

($AR(22)$).

Noticeably, for T^{DEV} the AR is at most 1 indicating very light dependence; together with the LB test with $\rho \geq 0.1424$, thus no evidence of stationarity at all. This allows us to confirm the independence of the observations. With T^{HOST}, AR is larger than 11, revealing stronger dependence between observations in the form of stationarity; LB has small ρ. Crucially for the applicability of EVT, the stronger stationarity of the T^{HOST} cases does not reflect into dependence of extreme observations, being the exponential trend of the ACFs with respect to lags. This is also supported by the extremogram results, in Figure 28. In there, the extremogram estimation $\hat{\rho}(h)$ varying lag h is represented. Small $\hat{\rho}$-values i.e., less than 0.05 suggest that the series has no dependences at the extremes. The extremal index θ confirms that, hence the resulting EVT pWCET estimation for T^{HOST} is equal to the one in case of full independence, Theorem 3, being $\theta \approx 1$.

The trends we could find in the measurement-bases distributions through the stationarity tests, therefore give us support to further statistically investigate measurements seeking the worst-case execution conditions.

The identical distribution, Kolmogorov-Smirnov (KS) test, is verified for all traces, with $\rho > 0.05$. It suffices to check if the observations follow the same distribution: indeed, this is always the case whenever the observations are taken with the same execution conditions.

To further comment on the different behaviour of T^{DEV} and T^{HOST} cases, notice the differences in Figure 26 and Figure 27. For T^{DEV}, the non stationarity (LB test and ACF residuals) is clearly explained with the trace of the standardised residuals: there is no evident pattern, thus it resembles white noise. In case of T^{HOST}, an execution pattern appears, more evident with VOR-32 T^{HOST}. The pattern is not that strong since ACF residuals and Ljung-Box outline stationarity until leg 5, VOR-32 T^{HOST}. Hence, it is not a strong stationarity, but stationarity is present anyway.

	T^{DEV}	T^{HOST}
VOR-1 runs (ρ)	0.3175	$5.235e-13$
VOR-8 runs (ρ)	0.7844	$< 2.2e-16$
VOR-28 runs (ρ)	0.664	$1.336-07$
VOR-32 runs (ρ)	0.5288	0.6189
VOR-1 KS (ρ)	0.9987	0.267
VOR-8 KS (ρ)	0.9601	0.532
VOR-28 KS (ρ)	0.6104	0.391
VOR-32 KS (ρ)	0.727811	0.5861
VOR-1 LB (ρ)	0.7407	$< 2.2e-16$
VOR-8 LB (ρ)	0.1424	$4.622e-07$
VOR-28 LB (ρ)	0.9205	$< 2.2e-16$
VOR-32 LB (ρ)	0.9715	$6.988e-05$
VOR-1 AR	1	26
VOR-8 AR	0	12
VOR-28 AR	0	22
VOR-32 AR	0	22
VOR-1 θ	1	1
VOR-8 θ	0.992	1
VOR-28 θ	1	1
VOR-32 θ	1	0.994

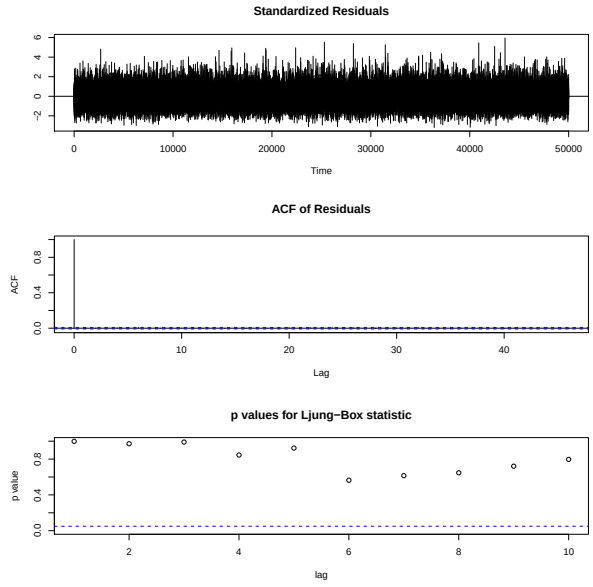
Table 1: Independence, stationarity and extremal tests.

With T^{HOST} we can see that there is no randomicity anymore, except for VOR-32 T^{HOST}. Moreover, execution peaks with a certain periodicity appear. Conversely, in case of T^{DEV} the appearances of peaks do not exhibit any periodic trend.

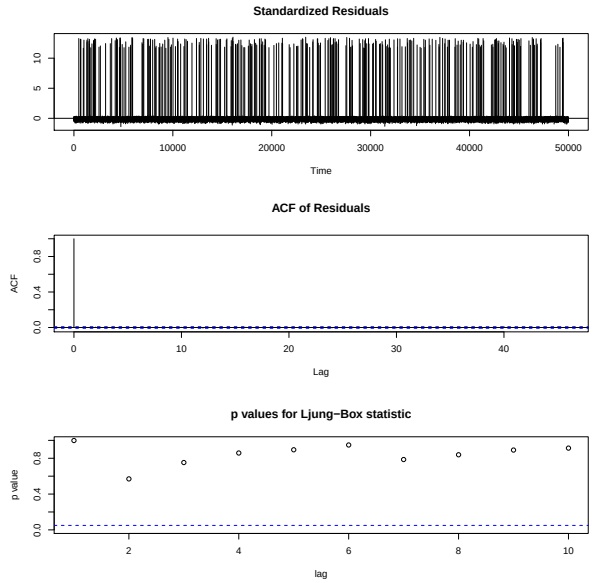
Seeking the worst case by investigating different execution conditions, we can see how the VOR-32, unsurprisingly, represents the worst-case among the ones considered (VOR-1, VOR-8, VOR-28, and VOR-32), being the case with larger observations. In Figure 29 we have represented the measurement-based distributions as Cumulative Distribution Functions (CDFs). In there we can also see that there is no measurable difference between VOR-28 and VOR-32 at both T^{DEV} and T^{HOST} cases.

7.3.2 From the Measurements to the pWCET

Finally we apply the EVT, in particular the block maxima version of the EVT [48]. In this chapter we do not give any detail about the complexity of the block maxima EVT



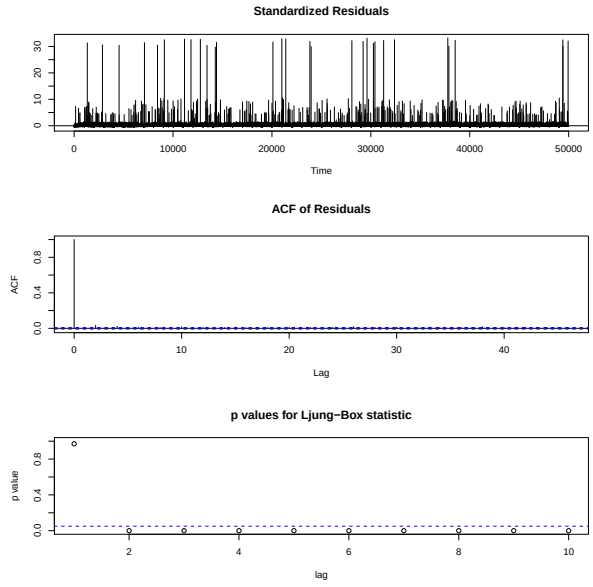
(a) VOR-1 T^{DEV}



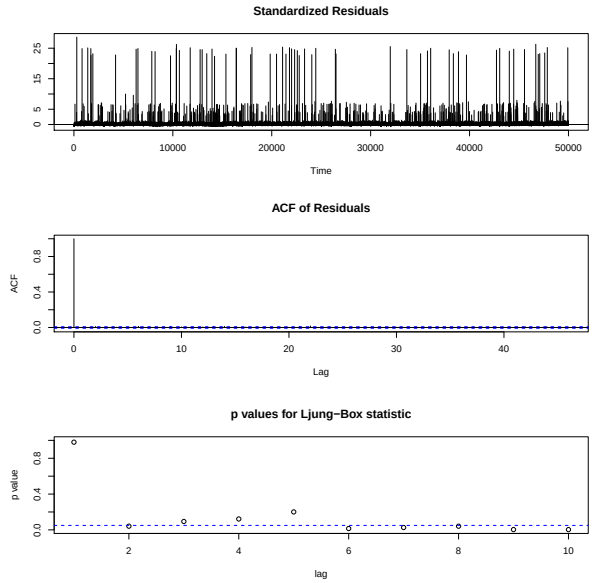
(b) VOR-32 T^{DEV}

Figure 26: Statistics from the autocorrelation function (ACF) and the Ljung-Box statistics. VOR-1 and VOR-32 T^{DEV} compared.

due to parameter decision (notably the block size), and we consider a block size of 25 observations. The application of the EVT is meant to compare the pWCETs of the different execution conditions. Figures 30 and 31 illustrate the differences accuracy in between VOR-x cases. The CDF representation is applied to the EVT pWCET



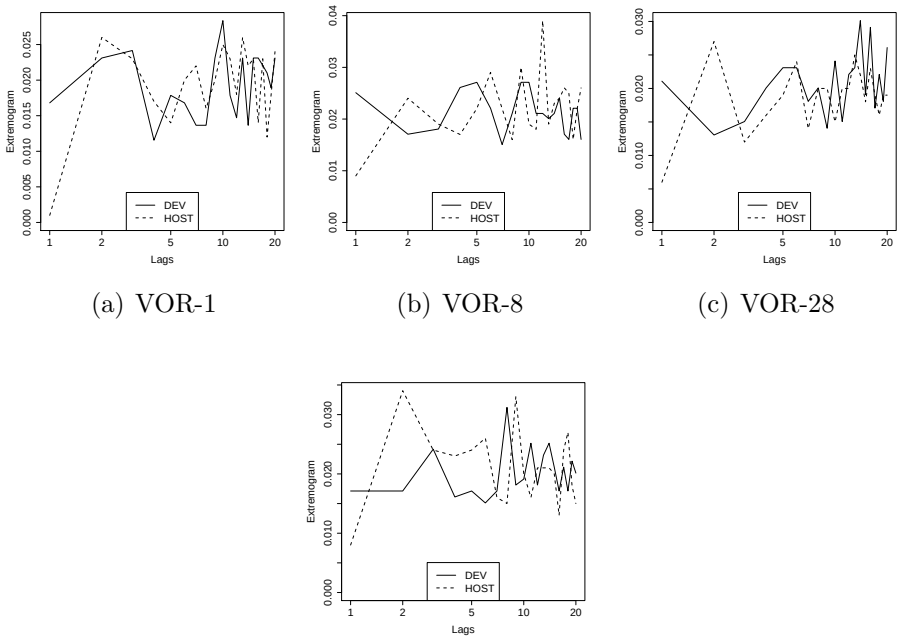
(a) VOR-1 T^{HOST}



(b) VOR-32 T^{HOST}

Figure 27: Statistics from the autocorrelation function (ACF) and the Ljung-Box statistics. VOR-1 and VOR-32 T^{HOST} compared.

distribution estimations. Although the real pWCET is not known, we can still reason about the accuracy of the pWCET estimations. For VOR-1 T^{DEV} and VOR-8 T^{DEV}, the EVT is closer to the measurements while for VOR-28 T^{DEV} and VOR-32 T^{DEV} it is less close. This is due to the shape of the measurement distributions. Wider

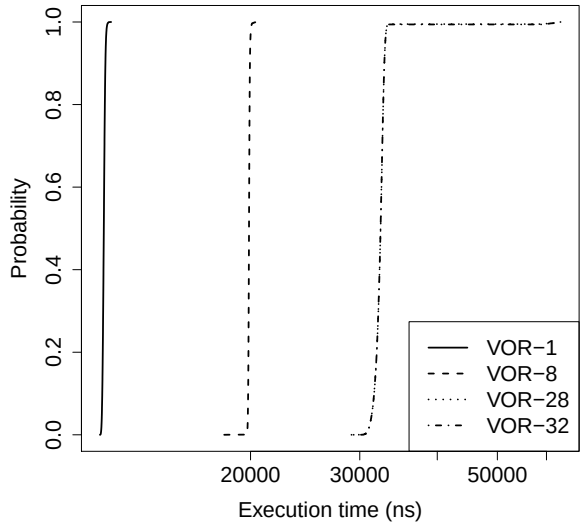


(a) VOR-1 (b) VOR-8 (c) VOR-28

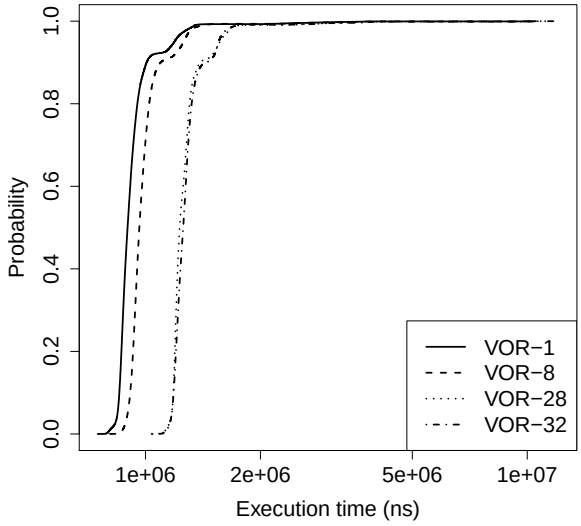
(d) VOR-32

Figure 28: Measurement extremogram up to 20 observations lag. T^{DEV} and T^{HOST} compared.

distributions (larger execution variability) means that rare events could be far away from the average behaviour. The EVT has to consider that in order to be safe: possibly much larger values than the measured ones have to be included. For the T^{HOST} cases, the measured distributions are consistently even wider, due to larger peaks on the execution times and two different peaks, visible in the residual representation of Figure 27. This makes the measured distributions resemble bi-variate distributions (Figure 31) and motivates the smaller estimation accuracy from the EVT. Table 2 shows the EVT estimations of the pWCET values at probability 10^{-6}, 10^{-9}, and 10^{-12} for both T^{DEV} and T^{HOST} cases; the probabilistic worst-case execution times are in ns. Those values are exceeding thresholds C, from the 1-CDF representation, and recall that the associated probabilities p are the probabilities of exceeding that threshold, $p(C) = P\{\mathcal{C}^* > C\}$ being $\mathcal{C}^*$ the EVT pWCET distribution estimation. These results illustrate the pWCET variation at different probability thresholds. To explain



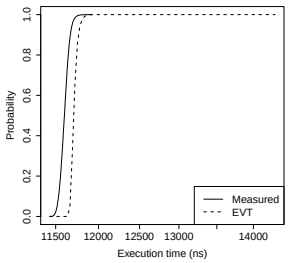
(a) T^{DEV}



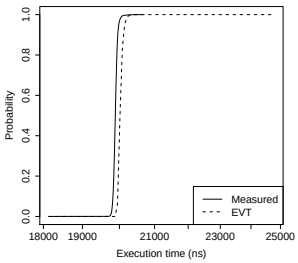
(b) T^{HOST}

Figure 29: Measurements for all the VORONOI cases. CDF representation of the distributions.

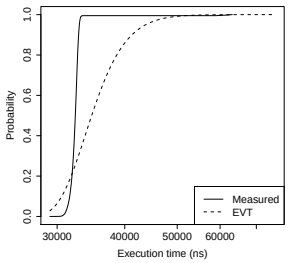
the large differences of the VOR-28 and VOR-32 T^{DEV} EVT estimations with respect to their measurements, again, we need to consider the variability of the measurement distributions: in order to be safe, with large variabilities and stationarity, the EVT looses accuracy. With narrow distributions like VOR-1 and VOR-8, the EVT can



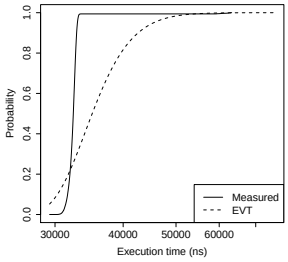
(a) VOR-1 T^{DEV}



(b) VOR-8 T^{DEV}



(c) VOR-28 T^{DEV}

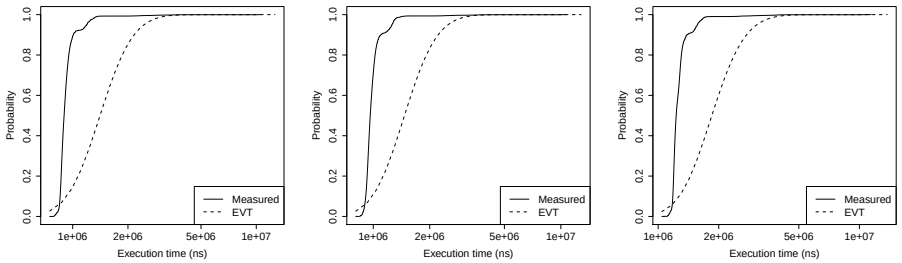


(d) VOR-32 T^{DEV}

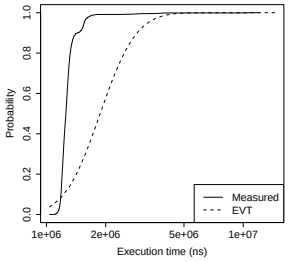
Figure 30: EVT applied to VOR-1, VOR-8, VOR-28 and VOR-32 T^{DEV}. Comparison of measurements vs EVT, CDF representations.

better model the measurements; the resulting pWCET estimations are closer to the observed execution times. For T^{HOST}, the poorest accuracy is due to the quality of the measured distribution. We also notice that the exceeding values for T^{HOST} for all VOR-x cases, for the same probability threshold, are of similar magnitude at each other. We conclude, empirically, that this is because T^{HOST} is dominated by the one-off costs of the CUDA driver execution and bus transfer launch, rather than the size of the problem instance (number of thread blocks). Indeed, $T^{HOST} \gg T^{DEV}$ in our experiments.

Figure 32 is to give informal evidence to EVT pWCET differences. Although the EVT provides the pWCET from a set of measurements $\mathcal{C}$, alone it is not enough to conclude about the task pWCET in any possible execution condition. Since the pWCET estimates for VOR-28 and VOR-32 (the cases with more thread blocks) are not inferable from those of VOR-1 and VOR-8, it is necessary to include the worst-case



(a) VOR-1 T^{HOST} (b) VOR-8 T^{HOST} (c) VOR-28 T^{HOST}



(d) VOR-32 T^{HOST}

Figure 31: EVT applied to VOR-1, VOR-8, VOR-28 and VOR-32 T^{HOST}. Comparison of measurements vs EVT, CDF representations.

execution condition (in terms of thread blocks) in order to guarantee safe pWCET estimations $\mathcal{C}^*$. Among the measurements made, VOR-32 is the worst-case for both T^{DEV} and T^{HOST}. The EVT statistical estimation out of the VOR-32 can provide the safety guarantee that real-time analyses require upper-bounding the pWCET estimation for all the other measurements.

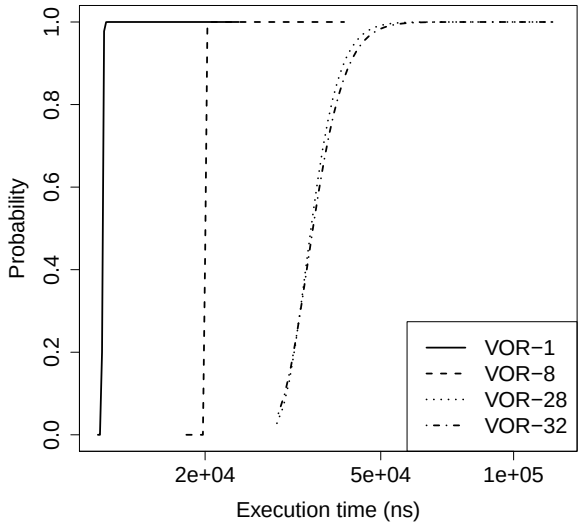
A few interesting observations on Table 2: (1) the gap between VOR-1 (1 thread block on 1 SM) and VOR-8 (8 thread blocks in parallel, on different SMs) quantifies the effect of contention across SMs for L2 and GPU main memory; (2) unlike the measured values, the EVT for VOR-8 and VOR-32 for a given probability, does not scale linearly with the thread blocks; (3) the almost identical VOR-28 and VOR-32 pWCET estimations are evidence of a balanced thread block assignment to SMs; the pWCET in VOR-28 (where some SMs get 3 and some get 4 thread blocks) is determined by those SMs with 4 thread blocks (same as all SMs in VOR-32).

	10^{-6}	10^{-9}	10^{-12}
VOR-1 T^{DEV}	12083	12278	12474
VOR-8 T^{DEV}	20600	20989	21248
VOR-28 T^{DEV}	80061	104613	128051
VOR-32 T^{DEV}	88252	115189	142510
VOR-1 T^{HOST}	6697335	9283349	12127964
VOR-8 T^{HOST}	6561744	9438101	12053516
VOR-28 T^{HOST}	8007463	11350140	14692817
VOR-32 T^{HOST}	8985862	12812711	16345188

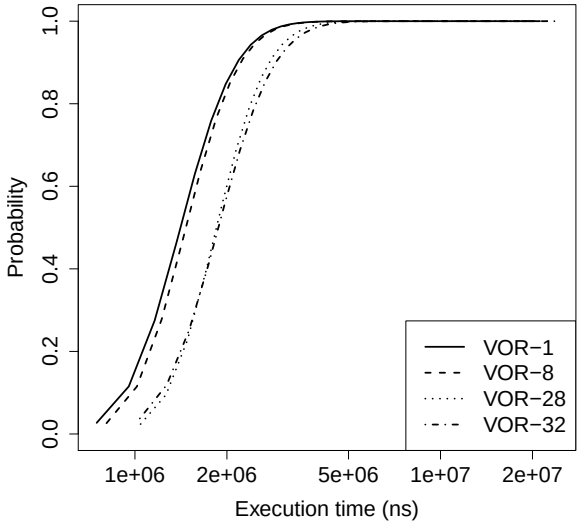
Table 2: EVT estimates for T^{DEV} and T^{HOST} at 10^{-6}, 10^{-9}, and 10^{-12} probability thresholds.

7.4 Summary

Through the work presented in this chapter we demonstrated that it is possible to apply a pWCET analysis approach based on measurements, statistic analysis, and EVT to parallel applications running on GPUs. We have proficiently extended applicability of EVT to less constraining hypotheses than independence. And that provides a way for obtaining accurate WCET estimates, for the desired confidence level, despite the lack of detailed public documentation on the GPU's memory subsystem and its internal scheduling.



(a) T^{DEV}



(b) T^{HOST}

Figure 32: CDF EVT distributions for T^{DEV}, T^{HOST}.

8 Conclusion

In the thesis statement presented in Chapter 1, we expressed our perspective about the potential of GPU timing analysis for the real-time systems domain. We supported that statement by developing GPU timing analysis approaches for real-time systems: optimization-based, metaheuristic-based and statistical measurement-based. Optimization-based and metaheuristic-based approaches represent a static branch of timing analysis subject to the theoretical model of GPU hardware that we proposed in Chapter 4. Our probabilistic measurement-based approach represents a statistical fork of a measurement-based branch of timing analysis. It allows us to target real hardware and demonstrates great potential for practical usage.

The strategy of our research was somehow similar to the breadth-first search in graphs. Instead of diving deep in one of the approaches, we opted on showing the big picture of the potential of GPU timing analysis. Thus, each of these approaches have some room for improvement. In the following, we discuss such improvements for optimization-based (Section 8.1), metaheuristic-based (Section 8.2), statistical measurement-based (Section 8.3) approaches, and finally conclude in Section 8.4.

8.1 On the optimization-based approach

First, in Chapter 5, the technique for computing a pessimistic upper bound on the worst-case makespan was presented. Then we used the outcome of this technique to formulate the optimization problem for finding an exact worst-case makespan and the corresponding schedule. Since the exact approach is computationally heavy for a large number of warps, we also introduced a simple way of obtaining, at only a fraction of the time, a safe estimate that is only marginally pessimistic subject to the simplifying assumptions.

The core of the optimization-based approach is the formulation of an optimization

problem that searches for the worst-case execution requirement. This formulation is also the main factor in terms of performance, thus, the efficiency of the problem solving determines the performance of the whole technique. The success of the formulation also depends on the model of the hardware and its configuration. Therefore, it would be interesting to analyze alternative scenarios for the sake of deriving some generic guidance for the application of the formulations presented in Chapter 5.

Another aspect of our theoretical model of GPU hardware, is that we addressed only a single streaming multiprocessor. However, since a GPU contains many streaming multiprocessors, an interesting problem to address is the extension of this approach to the case of a kernel execution over multiple streaming multiprocessors. Doing so will require faithfully modelling how warps are dispatched/partitioned among streaming multiprocessors – something which, to the best of our understanding, is either not fully documented at the moment or subject to change between revisions. On the other hand, adding some modelling of the memory subsystem would be crucial for making this approach to be more realistic. This would require serious efforts, taking into account the absence of publicly available information about the internal organization of the GPUs.

8.2 On the metaheuristic-based approach

In Chapter 6, an approach for obtaining a tight lower bound on the worst-case makespan was presented. This approach is based on the metaheuristic of simulated annealing and is capable of converging to a relatively tight estimate within short time. An important aspect is that the combination of the metaheuristic-based approach and the optimization-based approach provides both an upper-bound and a lower-bound on the worst-case makespan. This could be very beneficial for the cases when an exact solution cannot be found tractably.

As a next step, for additional confidence, and even though the degree of latency

hiding makes this less of an issue, it would be interesting to relax an optimistic aspect of the approach – the modelling of the memory subsystem and the absence of cache misses.

Similarly, since the GPU comprises multiple streaming multiprocessors, the extension of the approach to derive a makespan for GPU threads executing over the entire array of available streaming multiprocessors would be an interesting problem to address. Such an extension is not straightforward because the dispatching of the warps among streaming multiprocessors is undocumented as we already mentioned. Moreover, since multiprocessors within the GPU chip share the interconnection network, L2 cache and GPU main memory, there will be contention upon access to those resources. This contention needs to be modelled and accounted for by the analysis, even if the corresponding arbitration protocols are, likewise, undocumented.

8.3 On the statistical measurement-based approach

The output of the optimization-based approach and the metaheuristic-based approach was only safe subject to an *optimistic* assumption regarding cache misses, that was imposed due to control variable explosion in the first technique. The intension to work with real hardware processing real-life GPU applications brought us to the approach presented in Chapter 7, which uses measurements of end-to-end execution but which, through Statistical Analysis and Extreme Value Theory, can "predict" worst-case timing behavior even when that is not observable in the high-water mark times. In using measurements, we also largely sidestep the lack of public knowledge about the characteristics of the memory subsystem, which hampered us in the approaches discussed Chapter 5 and Chapter 6.

We believe that the statistical measurement-based approach has a great potential for practical use. We intend to continue the GPU probabilistic timing analysis investigating other system configurations and/or other system elements. For exam-

ple, by also considering different shared memory/L1 configurations (16/48 or 48/16 KB) rather than just different input sizes. The sensitivity analysis will be applied to system configurations and system parameters to evaluate their effect on the pWCET estimates. This will give us the possibility to develop an aided-design probabilistic framework for more deterministic GPU development.

8.4 Closing remarks

When we started this research, the GPU computing ecosystems did not include integrated GPUs. However, these days, the integrated GPUs are already available for GPU computing. We expect the rapid growth of their popularity and in terms of timing analysis this trend looks very promising. The plans on the integration of GPUs with other chips on the same die and the addition of the 3D memory technology revealed by the chip-makers might decrease the data-transfer latencies and make the hardware more amenable to the analysis in some aspects. Thus, we believe that this line of work will also apply to the next generations of GPU architectures.

We also hope that the research discussed in this thesis contributes in promoting the GPUs in the domain of real-time systems and will facilitate the future efforts on GPU timing analysis.

Appendix

Lemma 1. $\forall i \quad 1 \leq i \leq I$ *such that* $i, I \in \mathbb{N}$*;* $I \geq 2$ *and* $x_i, X \in \{0, 1\}$*:*

If inequality

$$\frac{1}{I} \sum_{i=1}^{I} x_i \leq X \leq \sum_{i=1}^{I} x_i \tag{94}$$

is valid, then

$$X = \vee_{i=1}^{I} x_i$$

Proof. Let us consider two complementary cases:

$$\text{Case 1: } \forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 0 \tag{95}$$

$$\text{Case 2: } \exists j \quad 1 \leq j \leq I \quad j \in \mathbb{N} \quad x_j = 1 \tag{96}$$

In Case 1, from (95) it follows that $\sum_{i=1}^{I} x_i = 0$, which in turn means that $\frac{1}{I} \sum_{i=1}^{I} x_i = 0$. Then according to (94), $0 \leq X \leq 0$ which means that $X = 0$. But from the assumption of the case, it also holds that $\vee_{i=1}^{I} x_i = 0$ – therefore $X = \vee_{i=1}^{I} x_i$.

In Case 2, from (96) it follows that $\sum_{i=1}^{I} x_i \geq 1$ and therefore $\frac{1}{I} \sum_{i=1}^{I} x_i > 0$. Combining this with the (94) and the fact that $X \in \{0, 1\}$, we obtain that $X = 1$. Additionally, $as \vee_{i=1}^{I} x_i = 1$, therefore, also in this case, $X = \vee_{i=1}^{I} x_i$.

Therefore, in all cases, $X = \vee_{i=1}^{I} x_i$. □

Lemma 2. $\forall i \quad 1 \leq i \leq I$ *such that* $i, I \in \mathbb{N}$*;* $I \geq 2$ *and* $x_i, X \in \{0, 1\}$

If

$$X = \vee_{i=1}^{I} x_i$$

then inequality (94)

$$\frac{1}{I} \sum_{i=1}^{I} x_i \leq X \leq \sum_{i=1}^{I} x_i$$

is valid.

Proof. Again, we explore two complementary cases:

$$\text{Case 1: } \forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 0 \tag{97}$$

$$\text{Case 2: } \exists j \quad 1 \leq j \leq I \quad j \in \mathbb{N} \quad x_j = 1 \tag{98}$$

In Case 1, from (97) follows that $\sum_{i=1}^{I} x_i = 0$ and consequently $\frac{1}{I} \sum_{i=1}^{I} x_i = 0$. According to definition of X and (97), $X = 0$. Therefore inequality (94) is valid in Case 1.

In Case 2, from (98) follows that $\sum_{i=1}^{I} x_i \geq 1$ and $\frac{1}{I} \sum_{i=1}^{I} x_i > 0$. According to definition of X and (98), $X = 1$. Therefore inequality (94) is valid for Case 2 as well.

Hence, in all cases, inequality (94) holds. □

Theorem 4. $\forall i \quad 1 \leq i \leq I$ *such that* $i, I \in \mathbb{N}$*;* $I \geq 2$ *and* $x_i, X \in \{0, 1\}$*:*

An inequality (94)

$$\frac{1}{I} \sum_{i=1}^{I} x_i \leq X \leq \sum_{i=1}^{I} x_i$$

is equivalent to the equality $X = \vee_{i=1}^{I} x_i$

Proof. Follows from Lemma 1 and Lemma 2. □

Lemma 3. $\forall i \quad 1 \leq i \leq I$ *such that* $i, I \in \mathbb{N}$*;* $I \geq 2$ *and* $x_i, X \in \{0, 1\}$

If inequality

$$-\frac{I-1}{I}+\frac{1}{I}\sum_{i=1}^{I}x_i \leq X \leq \frac{1}{I}\sum_{i=1}^{I}x_i \tag{99}$$

is valid, then

$$X = \wedge_{i=1}^{I} x_i$$

Proof. Let us consider two complementary cases:

$$\text{Case 1: } \forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 1 \tag{100}$$

$$\text{Case 2: } \exists j \quad 1 \leq j \leq I \quad j \in \mathbb{N} \quad x_j = 0 \tag{101}$$

In Case 1, from (100) it follows that $\sum_{i=1}^{I} x_i = I$ and consequently $\frac{1}{I}\sum_{i=1}^{I} x_i = 1$, $-\frac{I-1}{I}+\frac{1}{I}\sum_{i=1}^{I} x_i = \frac{1}{I} > 0$. Via substitution to (99) we then obtain $0 < X \leq 1$, which means that $X = 1$. Additionally, it holds that $\wedge_{i=1}^{I} x_i = 1$ – therefore $X = \wedge_{i=1}^{I} x_i$.

In Case 2, from (101) it follows that $\sum_{i=1}^{I} x_i < I$ and consequently $0 \leq \frac{1}{I}\sum_{i=1}^{I} x_i < 1$, $-\frac{I-1}{I}+\frac{1}{I}\sum_{i=1}^{I} x_i \leq 0$. Via substitution to (99) we obtain $0 \leq X < 1$, which means that $X = 0$. Additionally it holds that $\wedge_{i=1}^{I} x_i = 0$ – therefore $X = \wedge_{i=1}^{I} x_i$.

Therefore, in all cases, $X = \wedge_{i=1}^{I} x_i$. □

Lemma 4. $\forall i \quad 1 \leq i \leq I$ *such that* $i, I \in \mathbb{N}$*;* $I \geq 2$ *and* $x_i, X \in \{0, 1\}$

If

$$X = \wedge_{i=1}^{I} x_i$$

then inequality (99)

$$-\frac{I-1}{I}+\frac{1}{I}\sum_{i=1}^{I}x_i \leq X \leq \frac{1}{I}\sum_{i=1}^{I}x_i$$

is valid.

Proof. Let us consider two complementary cases:

$$\text{Case 1: } \forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 1 \tag{102}$$

$$\text{Case 2: } \exists j \quad 1 \leq j \leq I \quad j \in \mathbb{N} \quad x_j = 0 \tag{103}$$

In Case 1, from (102) it follows that $X = 1$, $-\frac{I-1}{I} + \frac{1}{I}\sum_{i=1}^{I} x_i = \frac{1}{I} < 1$, and $\frac{1}{I}\sum_{i=1}^{I} x_i = 1$. Therefore (99) in this case is valid.

In Case 2, from (103) it follows that $X = 0$, $-\frac{I-1}{I} + \frac{1}{I}\sum_{i=1}^{I} x_i \leq 0$, and $0 \leq \frac{1}{I}\sum_{i=1}^{I} x_i \leq 1$. Therefore (99) is valid for this case as well.

Therefore inequality (99) holds in all cases. □

Theorem 5. $\forall i \quad 1 \leq i \leq I$ *such that* $i, I \in \mathbb{N}$*;* $I \geq 2$ *and* $x_i, X \in \{0, 1\}$

The inequality (99)

$$-\frac{I-1}{I} + \frac{1}{I}\sum_{i=1}^{I} x_i \leq X \leq \frac{1}{I}\sum_{i=1}^{I} x_i$$

is equivalent to the equality

$$X = \wedge_{i=1}^{I} x_i$$

Proof. Follows from Lemma 3 and Lemma 4. □

Lemma 5. $\forall i \quad 1 \leq i \leq I$ *such that* $i, I \in \mathbb{N}$*;* $I \geq 2$ *and* $x_i, y, Z \in \{0, 1\}$*:*

If inequality

$$\frac{1}{2} \times (-\frac{I-1}{I} + \frac{1}{I} \times \sum_{i=1}^{I} x_i + y) - \frac{1}{2 \times I} < Z \leq \frac{1}{I} \times \sum_{i=1}^{I} x_i + y \tag{104}$$

is valid, then

$$Z = (\wedge_{i=1}^{I} x_i) \vee y \tag{105}$$

Proof. For the sake of brevity, we denote the left-hand expression and the right-hand expression of the double inequality (104) as L and R respectively:

$$L = \frac{1}{2} \times (-\frac{I-1}{I} + \frac{1}{I} \times \sum_{i=1}^{I} x_i + y) - \frac{1}{2 \times I} \tag{106}$$

$$R = \frac{1}{I} \times \sum_{i=1}^{I} x_i + y \tag{107}$$

Let us consider two complementary cases:

$$\text{Case 1: } \forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 1 \tag{108}$$

$$\text{Case 2: } \exists j \quad 1 \leq j \leq I \quad j \in \mathbb{N} \quad x_j = 0 \tag{109}$$

In Case 1, from (108) it follows that $\sum_{i=1}^{I} x_i = I$ and consequently

$$\frac{1}{I} \sum_{i=1}^{I} x_i = 1 \tag{110}$$

Hence, from Equation (106)

$$L = \frac{1}{2} \times (-\frac{I-1}{I} + 1 + y) - \frac{1}{2 \times I} = \frac{1}{2} \times (\frac{-I+1+I}{I} + y) - \frac{1}{2 \times I} = \frac{1}{2} \times (\frac{1}{I} + y) - \frac{1}{2 \times I}$$

Therefore,

$$\begin{aligned} &\forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 1 \\ &L = \frac{1}{2} \times (\frac{1}{I} + y) - \frac{1}{2 \times I} \end{aligned} \tag{111}$$

From Equation (107) and Equation (110) we get

$$R = 1 + y \tag{112}$$

We can substitute the left-hand side and the right-hand side of the double inequality (104) with the right-hand sides of Equation (111) and Equation (112) respectively:

$$\begin{aligned} &\forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 1 : \\ &\frac{1}{2} \times (\frac{1}{I} + y) - \frac{1}{2 \times I} < Z \leq 1 + y \end{aligned} \tag{113}$$

Inside Case 1, we can consider two complementary subcases:

$$\begin{aligned} &\forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 1 \\ &\text{Case 1.0:} \quad y = 0 \\ &\text{Case 1.1:} \quad y = 1 \end{aligned} \tag{114}$$

In Case 1.0, we can rewrite Equation (113) by substituting y with 0:

$$\begin{aligned} \frac{1}{2} \times (\frac{1}{I} + 0) - \frac{1}{2 \times I} < Z \leq 1 + 0 \quad & \iff \\ 0 < Z \leq 1 \quad & (115) \end{aligned}$$

By the definition, Z is a binary value $Z \in \{0, 1\}$, therefore, Equation (115) specifies that $Z = 1$. Notice, that

$$\begin{aligned} & \forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 1, \quad y = 0 \\ & (\wedge_{i=1}^{I} x_i) \vee y = 1 \vee 0 = 1 \quad (116) \end{aligned}$$

Therefore, in Case 1.0, $Z = (\wedge_{i=1}^{I} x_i) \vee y = 1$ and Lemma 5 is valid.

In Case 1.1, we substitute y with 1 in Equation (113):

$$\begin{aligned} \frac{1}{2} \times (\frac{1}{I} + 1) - \frac{1}{2 \times I} < Z \leq 1 + 1 \quad & \iff \\ \frac{I+1}{2 \times I} - \frac{1}{2 \times I} < Z \leq 2 \quad & \iff \\ \frac{1}{2} < Z \leq 2 \quad & (117) \end{aligned}$$

Since Z can have only two possible values, 0 or 1, Equation (117) specifies that $Z = 1$. Given that

$$\begin{aligned} & \forall i \quad 1 \leq i \leq I \quad i \in \mathbb{N} \quad x_i = 1, \quad y = 1 \\ & (\wedge_{i=1}^{I} x_i) \vee y = 1 \vee 1 = 1 \quad (118) \end{aligned}$$

$Z = (\wedge_{i=1}^{I} x_i) \vee y = 1$. Hence, in Case 1.1, Lemma 5 holds as well.

In Case 2, from Equation (109) we know that $0 \leq \sum_{i=1}^{I} x_i < I$ and consequently

$$0 \leq \frac{1}{I} \sum_{i=1}^{I} x_i < 1 \tag{119}$$

From Equation (106) and Equation (119) we get the following bounds on the left-hand expression of the double inequality (104) marked as L.

$$\begin{aligned} \frac{1}{2} \times (-\frac{I-1}{I} + 0 + y) - \frac{1}{2 \times I} \leq L < \frac{1}{2} \times (-\frac{I-1}{I} + 1 + y) - \frac{1}{2 \times I} \quad &\iff \\ \frac{1}{2} \times (-\frac{I-1}{I} + y) - \frac{1}{2 \times I} \leq L < \frac{1}{2} \times (\frac{-I+1+I}{I} + y) - \frac{1}{2 \times I} \quad &\iff \\ \frac{1}{2} \times (-\frac{I-1}{I} + y) - \frac{1}{2 \times I} \leq L < \frac{1}{2} \times (\frac{1}{I} + y) - \frac{1}{2 \times I} \end{aligned} \tag{120}$$

To construct the bounds for the right-hand expression of the double inequality (104) (marked as R) we use Equation (107) and Equation (119):

$$\begin{aligned} 0 + y \leq & R < 1 + y \\ y \leq & R < 1 + y \end{aligned} \tag{121}$$

Inside Case 2, we consider the following complementary subcases:

$$\begin{aligned} &\exists j \quad 1 \leq j \leq I \quad j \in \mathbb{N} \quad x_j = 0 \\ &\text{Case 2.0:} \quad y = 0 \\ &\text{Case 2.1:} \quad y = 1 \end{aligned} \tag{122}$$

In Case 2.0, we substitute y with its value 0 in Equation (120) to get the bounds on

the left-hand side L of the double inequality (104):

$$\begin{aligned} \frac{1}{2} \times (-\frac{I-1}{I} + 0) - \frac{1}{2 \times I} \leq L < \frac{1}{2} \times (\frac{1}{I} + 0) - \frac{1}{2 \times I} \quad &\iff \\ \frac{-I+1-1}{2 \times I} \leq L < \frac{1}{2 \times I} - \frac{1}{2 \times I} \quad &\iff \\ -\frac{1}{2} \leq L < 0 \quad &(123) \end{aligned}$$

For the right-hand side R of the double inequality (104), we substitute y with 0 in Equation (121):

$$\begin{aligned} 0 \leq & R < 1 + 0 \\ 0 \leq & R < 1 \quad (124) \end{aligned}$$

From Equation (123) $L \in [-\frac{1}{2}, 0)$ and from Equation (124) $R \in [0, 1)$ (see Figure 33). From the double inequality (104) $Z \in (L, R]$, hence its value should be somewhere on the right side of 0 (included) and on the left side of 1 (excluded). Given that by definition $Z \in \{0, 1\}$, the only value that meets for all the constraints is $Z = 0$. Notice, that the value $Z = 1$ violates Equation (124) ($R \in [0, 1)$) and the double inequality (104) ($Z \in (L, R]$).

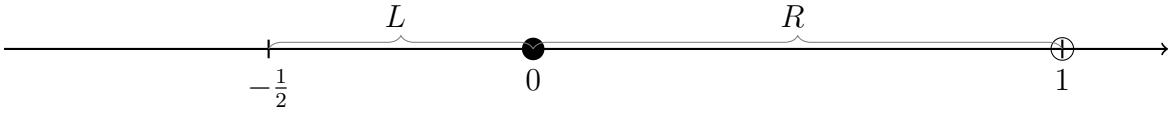


Figure 33: Determining the value of Z in Case 2.0

By checking Equation (105):

$$\begin{aligned} &\exists j \quad 1 \leq j \leq I \quad j \in \mathbb{N} \quad x_j = 0, \quad y = 0 \\ &(\wedge_{i=1}^{I} x_i) \vee y = 0 \vee 0 = 0 \end{aligned} \tag{125}$$

$Z = \wedge_{i=1}^{I} x_i) \vee y = 0$, hence Lemma 5 holds in Case 2.0.

In Case 2.1, y is substituted with 1. By doing this in Equation (120) we get the bounds for the left-hand side L of the double inequality (104):

$$\begin{aligned} \frac{1}{2} \times (-\frac{I-1}{I} + 1) - \frac{1}{2 \times I} \leq L < \frac{1}{2} \times (\frac{1}{I} + 1) - \frac{1}{2 \times I} \quad &\iff \\ \frac{-I+1+I}{2 \times I} - \frac{1}{2 \times I} \leq L < \frac{I+1}{2 \times I} - \frac{1}{2 \times I} \quad &\iff \\ 0 \leq L < \frac{1}{2} \end{aligned} \tag{126}$$

For the right-hand side R of the double inequality (104) we substitute y with 1 in Equation (121):

$$\begin{aligned} y \leq &R < 1 + y \\ 1 \leq &R < 2 \end{aligned} \tag{127}$$

From Equation (126) $L \in [0, \frac{1}{2})$ and from Equation (127) $R \in [1, 2)$ (see Figure 34). According to the double inequality (104) $Z \in (L, R]$, the value of Z has to be somewhere on the right side of $\frac{1}{2}$ (excluded) and on the left side of 1 (included). By definition, Z can be either 0 or 1, therefore, $Z = 1$ is the only value that can satisfy all the constraints. Notice, that the value $Z = 0$ is not eligible since it violates Equation (126) ($L \in [0, \frac{1}{2})$) and the double inequality (104) ($Z \in (L, R]$).

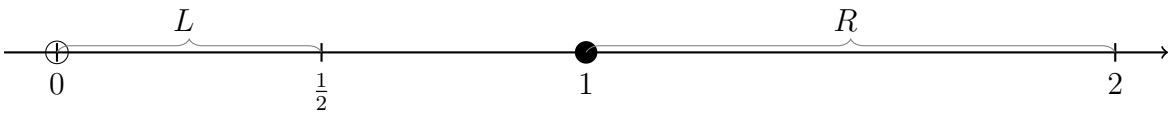


Figure 34: Determining the value of Z in Case 2.1

Let us check Equation (105):

$$\exists j \quad 1 \leq j \leq I \quad j \in \mathbb{N} \quad x_j = 0, \quad y = 1$$
$$(\wedge_{i=1}^{I} x_i) \vee y = 0 \vee 1 = 1 \tag{128}$$

$Z \in \{0, 1\}$, from double inequality (104) $Z \in (L, R]$ Hence, Lemma 5 is also valid in Case 2.1.

Thus, we showed that Lemma 5 holds in all subcases within Case 1 and Case 2. □

Lemma 6. *$\forall i \quad 1 \leq i \leq I$ such that $i, I \in \mathbb{N}$; $I \geq 2$ and $x_i, y, Z \in \{0, 1\}$:*

If the equality (105)

$$Z = (\wedge_{i=1}^{I} x_i) \vee y$$

holds, then the inequality (104)

$$\frac{1}{2} \times (-\frac{I-1}{I} + \frac{1}{I} \times \sum_{i=1}^{I} x_i + y) - \frac{1}{2 \times I} < Z \leq \frac{1}{I} \times \sum_{i=1}^{I} x_i + y$$

is valid.

Proof. Let us consider the boolean expression $(\wedge_{i=1}^{I} x_i) \vee y$ in the right-hand side of Equation (105) as a disjunction of the boolean expression $\wedge_{i=1}^{I} x_i$ and the binary variable y, for the sake of applying Theorem 4. In the formulation of the theorem, we substitute X with Z and $I = 2$ terms of the disjunction $x_1,\ x_2$ with $\wedge_{i=1}^{I} x_i$ and

y, hence the following bounds on the boolean expression $(\wedge_{i=1}^{I} x_i) \vee y$ are derived:

$$\frac{1}{2} \times \Big((\wedge_{i=1}^{I} x_i) + y\Big) \leq Z \leq (\wedge_{i=1}^{I} x_i) + y \tag{129}$$

Notice, that according to Theorem 5

$$-\frac{I-1}{I} + \frac{1}{I}\sum_{i=1}^{I} x_i \leq \wedge_{i=1}^{I} x_i \leq \frac{1}{I}\sum_{i=1}^{I} x_i \tag{130}$$

subject to the substitution of X with $\wedge_{i=1}^{I} x_i$ in the formulation of the theorem.

Let us consider the left-hand inequality of the double inequality (129)

$$\frac{1}{2} \times \Big((\wedge_{i=1}^{I} x_i) + y\Big) \leq Z \tag{131}$$

and the left-hand inequality of the double inequality (130)

$$-\frac{I-1}{I} + \frac{1}{I}\sum_{i=1}^{I} x_i \leq \wedge_{i=1}^{I} x_i \tag{132}$$

The right-hand side of Equation (132) appears in the left-hand side of Equation (131). Therefore, we can substitute the right-hand side of Equation (132) into the left-hand side of Equation (131):

$$\begin{aligned} \frac{1}{2} \times (-\frac{I-1}{I} + \frac{1}{I}\sum_{i=1}^{I} x_i + y) \leq \frac{1}{2} \times \Big((\wedge_{i=1}^{I} x_i) + y\Big) \leq & Z \qquad \Longleftrightarrow \\ \frac{1}{2} \times (-\frac{I-1}{I} + \frac{1}{I}\sum_{i=1}^{I} x_i + y) \leq & Z \end{aligned} \tag{133}$$

By subtracting a positive number $(\frac{1}{2\times I})$ from the left-hand side of Equation (133) we

can make the corresponding non-strict inequality strict:

$$\frac{1}{2} \times (-\frac{I-1}{I} + \frac{1}{I}\sum_{i=1}^{I} x_i + y) - \frac{1}{2 \times I} < Z \tag{134}$$

Let us consider the right-hand inequality of the double inequality (129)

$$Z \leq (\wedge_{i=1}^{I} x_i) + y \tag{135}$$

and the right-hand inequality of the double inequality (130)

$$\wedge_{i=1}^{I} x_i \leq \frac{1}{I}\sum_{i=1}^{I} x_i \tag{136}$$

The right-hand side of Equation (136) can be found in the left-hand side of Equation (135). Thus, we substitute the right-hand side of Equation (136) into the left-hand side of Equation (135).

$$Z \leq (\wedge_{i=1}^{I} x_i) + y \leq \frac{1}{I}\sum_{i=1}^{I} x_i + y \qquad \Longleftrightarrow$$

$$Z \leq \frac{1}{I}\sum_{i=1}^{I} x_i + y \tag{137}$$

One can combine the inequality (134) and the inequality (137) into a double inequality

$$\frac{1}{2} \times (-\frac{I-1}{I} + \frac{1}{I}\sum_{i=1}^{I} x_i + y) - \frac{1}{2 \times I} < Z \leq \frac{1}{I}\sum_{i=1}^{I} x_i + y \tag{138}$$

which is exactly the same as Equation (104) in the formulation of Lemma 6. □

Theorem 6. $\forall i \quad 1 \leq i \leq I$ *such that* $i, I \in \mathbb{N}$*;* $I \geq 2$ *and* $x_i, y, Z \in \{0, 1\}$*:*

The inequality (104)

$$\frac{1}{2} \times (-\frac{I-1}{I} + \frac{1}{I} \times \sum_{i=1}^{I} x_i + y) - \frac{1}{2 \times I} < Z \leq \frac{1}{I} \times \sum_{i=1}^{I} x_i + y$$

is equivalent to the equality (105)

$$Z = (\wedge_{i=1}^{I} x_i) \vee y$$

Proof. Follows from Lemma 5 and Lemma 6. □

Bibliography

[1] Advanced Micro Devices, Inc. ATI Stream Computing Programming Guide. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_CAL_Programming_Guide_v2.0.pdf, 2010.

[2] Charles Alexander and Matthew Sadiku. *Fundamentals of Electric Circuits*. McGraw-Hill, 2004.

[3] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

[4] Alia Atlas and Azer Bestavros. Statistical rate monotonic scheduling. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 123–132. IEEE, 1998.

[5] N. C. Audsley and K. Bletsas. Realistic analysis of limited parallel software / hardware implementations. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS2004)*, Toronto, Canada, may 2004. IEEE Computer Society, IEEE.

[6] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[7] Luis I Bacayo, John R Nickolls, and Bryon S Nordquist. Parallel data processing systems and methods using cooperative thread arrays and simd instruction issue, 2009. US Patent 7,584,342.

[8] J Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613 – 641, August 1978.

[9] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. *SIGPLAN Not.*, 45(5):105–114, January 2010.

[10] Sara S. Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-mei W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2012.

[11] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Proc. IEEE ISPASS*, 2009.

[12] James Balfour. CUDA threads and atomics. Lecture slides (2011) – http://mc.stanford.edu/cgi-bin/images/3/34/Darve_cme343_cuda_3.pdf.

[13] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[14] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *rtss*, pages 119–128. IEEE, 2007.

[15] Sanjoy Baruah. The federated scheduling of constrained-deadline sporadic dag task systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1323–1328. EDA Consortium, 2015.

[16] Sanjoy Baruah. Federated scheduling of sporadic dag task systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 179–186. IEEE, 2015.

[17] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global edf scheduling of systems of conditional sporadic dag tasks. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 222–231. IEEE, 2015.

[18] Sunandan Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 63–72. IEEE, 2012.

[19] Swagato Basumallick and Kelvin Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, volume 5. Citeseer, 1994.

[20] Mikhail Bautin, Ashok Dwarakinath, and Tzi cker Chiueh. Graphics Engine Resource Management. In *Proc. 15th ACM/SPIE Multimedia Computing and Networking Conference (MMCN)*, 2008.

[21] Jan Beirlant, Yuri Goegebeur, Johan Segers, and Jozef Teugels. *Statistics of extremes: theory and applications*. John Wiley & Sons, 2006.

[22] K. Berezovskyi, K. Bletsas, and B. Andersson. Makespan computation for GPU threads running on a single streaming multiprocessor. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, 2012.

[23] K. Berezovskyi, K. Bletsas, and Stefan M. Petters. Faster makespan estimation for GPU threads on a single streaming multiprocessor. In *Proc. ETFA*, 2013.

[24] Christoph Berg, Jakob Engblom, and Reinhard Wilhelm. Requirements for and design of a processor with predictable timing. In *Perspectives Workshop: Design of Systems with Predictable Behaviour, 16.-19. November 2003, volume 03471 of Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl*, 2004.

[25] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices*, volume 41, pages 158–168. ACM, 2006.

[26] Guillem Bernat, Anotione Colin, and Stefan M Petters. WCET analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 279–288. IEEE, 2002.

[27] Vandy Berten, Sébastien Collette, and Joël Goossens. Feasibility test for multi-phase parallel real-time jobs. In *Proceedings of the Work-in-Progress session of the IEEE Real-Time Systems Symposium*, volume 2009, pages 33–36. Citeseer, 2009.

[28] Adam Betts and Alastair F. Donaldson. Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *Proc. 25th ECRTS*, pages 193–202, 2013.

[29] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 447 –456, dec. 2009.

[30] K. Bletsas and N.C. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*, pages 525 – 531, aug. 2005.

[31] Konstantinos Bletsas, Neil Audsley, Wen-Hung Huang, Jian-Jia Chen, and Geoffrey Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical Report CISTER-TR-150713, CISTER-Research Centre in Realtime and Embedded Computing Systems, 2015.

[32] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic dag task model. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 225–233. IEEE, 2013.

[33] James V Bradley. Distribution-free statistical tests. 1968.

[34] Alan Burns and Andrew J Wellings. *Real-time systems and programming languages*, volume 2097. Addison-Wesley, 2010.

[35] José V Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro Gil, and Andy Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 204–212. IEEE, 1996.

[36] John Catsoulis. *Designing embedded hardware*. " O'Reilly Media, Inc.", 2005.

[37] F.J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. PROARTIS: Probabilistically analysable real-time systems. *ACM Transactions on Embedded Computing Systems*, 2011.

[38] V. Chavez-Demoulin and A.C. Davison. Modelling time series extremes. *REVSTAT*, 10(1), 2012.

[39] Jian-Jia Chen, Geoffrey Nelissen, and Wen-Hung Huang. A note to consider self-suspending as blocking. Technical report, Available at https://github.com/jjchentw/A-Note-to-Consider-Self-Suspending-as-Blocking, 2015.

[40] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Michael González Harbour, Neil Audsley, Raj Rajkumar, and Dionisio de Niz. Many Suspensions, Many Problems: A Review of Self-Suspending Tasks in Real-Time Systems. Technical report, 2016.

[41] Michael R. Chernick. A limit theorem for the maximum of autoregressive processes with uniform marginal distributions. *The Annals of Probability*, 9(1):145–149, 02 1981.

[42] Jen-Yao Chung, Jane WS Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *Computers, IEEE Transactions on*, 39(9):1156–1174, 1990.

[43] Krishna C.M. and Shin K.G. *Real-Time Systems*. McGraw-Hill, 1997.

[44] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: a parallel functional simulator for gpgpu. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 351–360. IEEE, 2010.

[45] Sébastien Collette, Liliana Cucu, and Joël Goossens. Algorithm and complexity for the global scheduling of sporadic tasks on multiprocessors with work-limited parallelism. *RTNS07*, page 123, 2007.

[46] Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, 2008.

[47] Melvin E Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, fall joint computer conference*, pages 139–146. ACM, 1963.

[48] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F. J. Cazorla. Measurement-Based

Probabilistic Timing Analysis for Multi-path Programs. In *Proc. 23nd ECRTS*, 2012.

[49] Richard A. Davis and Thomas Mikosch. The extremogram: A correlogram for extreme events. *Bernoulli Society for Mathematical Statistics and Probability*, 2009.

[50] Robert Davis, Luca Santinelli, Sebastian Altmeyer, Claire Maiza, and Liliana Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 168–179. IEEE, 2013.

[51] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[52] J.L Díaz, D.F. Garcia, K. Kim, C.G. Lee, L.L. Bello, López J.M., and O. Mirabella. Stochastic analysis of periodic real-time systems. In *23rd RTSS*, pages 289–300, 2002.

[53] Stewart Edgar and Alan Burns. Statistical analysis of wcet for scheduling. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 215–224. IEEE, 2001.

[54] Rudolf Eigenmann and David J Lilja. Von neumann computers. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1998.

[55] G. Elliott, B. Ward, and J. Anderson. Gpusync: Architecture-aware management of gpus for predictable multi-gpu real-time systems. In submission.

[56] Glenn A Elliott. *Real-time scheduling for GPUS with applications in advanced automotive systems*. PhD thesis, University of North Carolina at Chapel Hill, 2015.

[57] Paul Embrechts, Thomas Mikosch, and Claudia Klüppelberg. *Modelling Extremal Events: For Insurance and Finance*. Springer-Verlag, London, UK, UK, 1997.

[58] Frédéric Fauberteau, Serge Midonnet, and Manar Qamhieh. Partitioned scheduling of parallel real-time tasks on multiprocessor systems. *ACM SIGBED Review*, 8(3):28–31, 2011.

[59] Dror G Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[60] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(9):1627–1645, 2010.

[61] W Feng. Types of barrier synchronization, April 2009.

[62] Nathan Fisher, Sanjoy Baruah, and Theodore P Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 10–pp. IEEE, 2006.

[63] Nathan Fisher, Joël Goossens, and Sanjoy Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2):26–71, 2010.

[64] José Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luis Miguel Pinho. A multi-dag model for real-time parallel applications with conditional execution. In *The 30th ACM/SIGAPP Symposium On Applied Computing*, 2015.

[65] Steven Fortune. A sweepline algorithm for Voronoi diagrams. In *Proceedings of the second annual symposium on computational geometry*, pages 313 – 322, 1986.

[66] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data transfer matters for gpu computing. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 275–282. IEEE, 2013.

[67] Mark K Gardner and Jane WS Liu. Analyzing stochastic fixed-priority real-time systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 44–58. Springer, 1999.

[68] M Garrido and J Diebolt. The et test, a goodness-of-fit test for the distribution tail. In *Methodology, Practice and Inference, second international conference on mathematical methods in reliability*, pages 427–430, 2000.

[69] Vladimir Glavtchev, Pnar Muyan-Özçelik, Jeffrey M. Ota, and John D. Owens. Feature-based speed limit sign detection using a graphics processing unit. In *Intelligent Vehicles Symposium (IV)*, pages 195–200, 2011.

[70] Boris Gnedenko. Sur la distribution limite du terme maximum d'une serie aleatoire. *Annals of mathematics*, pages 423–453, 1943.

[71] Steve Goddard. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 60–71. IEEE, 1997.

[72] Steve Goddard. *On the Management of Latency in the synthesis of real-time signal processing systems from processing graphs*. PhD thesis, 1998.

[73] Dominik Göddeke. GPGPU:basic math tutorial. http://www.mathematik.uni-dortmund.de/

[74] Joël Goossens and Vandy Berten. Gang FTP scheduling of periodic and parallel rigid real-time tasks. In *18th International Conference on Real-Time and Network Systems*, pages 189–196, 2010.

[75] M. Gouiffès, A. Patri, and M. Vasiliu. Robust obstacles detection and tracking using disparity for car driving assistance. In *SPIE*, volume 7539, 2010.

[76] GPGPU. General-Purpose Computation on Graphics Hardware. http://www.gpgpu.org.

[77] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[78] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326, 1979.

[79] David Griffin and Alan Burns. Realism in statistical analysis of worst case execution times. In *WCET*, pages 44–53, 2010.

[80] E.J. Gumbel. *Statistics of Extremes*. Columbia University Press, 1958.

[81] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. *WCET*, 15:136–146, 2010.

[82] Ching-Chih Han and Kwei-Jay Lin. Scheduling parallelizable jobs on multiprocessors. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 59–67. IEEE, 1989.

[83] J. Hansen, S. Hissam, and G. A. Moreno. Statistical-based wcet estimation and validation. In *the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[84] Jeffery Hansen, Scott A Hissam, and Gabriel A Moreno. Statistical-based wcet estimation and validation. In *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[85] Damien Hardy and Isabelle Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 35–44. ACM, 2013.

[86] John Hayes. *Introduction to Digital Logic Design*. Addison Wesley, 1993.

[87] Christopher A. Healy. Integrating the timing analysis of pipelining and instruction caching. In *In IEEE Real-Time Systems Symposium*, pages 288–297, 1995.

[88] Manato Hirabayashi, Shinpei Kato, Masato Edahiro, Kazuya Takeda, Taiki Kawano, and Seiichi Mita. Gpu implementations of object detection using hog features and deformable models. In *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2013 IEEE 1st International Conference on*, pages 106–111. IEEE, 2013.

[89] Vesa Hirvisalo and Heiko Falk. On static timing analysis of gpu kernels. In *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39, pages 43–52. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[90] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163, 2009.

[91] Sunpyo Hong and Hyesoon Kim. A memory-level and thread-level parallelism aware GPU architecture performance analytical model. In *36th International Symposium on Computer Architecture (ISCA-36)*, June 2009.

[92] Tailen Hsing. On tail index estimation using dependent data. *The Annals of Statistics*, 1991.

[93] Wen-Hung Huang and Jian-Jia Chen. Schedulability and priority assignment for multi-segment self-suspending real-time tasks under fixed-priority scheduling. Technical report, Technical report, Dortmund, Germany, 2015.

[94] Wen-Hung Huang and Jian-Jia Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In *Design, Automation and Test in Europe (DATE), 2016*, 2016.

[95] IBM Corporation. IBM ILOG CPLEX Optimization Studio. Product brief – http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/modeling/, 2011.

[96] Kevin Jeffay and David Bennett. A rate-based execution abstraction for multimedia computing. In *Network and Operating Systems Support for Digital Audio and Video*, pages 64–75. Springer, 1995.

[97] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 304–314. IEEE, 1999.

[98] DJ Kaplan et al. Processing graph method specification version 1.0. *Unpublished Memorandum, The Naval Research Laboratory, Washington DC*, 1987.

[99] S. Kato and Y. Ishikawa. Gang edf scheduling of parallel task systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 459–468. IEEE, 2009.

[100] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 249 –258, july 2009.

[101] Shinpei Kato. Implementing open-source CUDA runtime. http://www.ertl.jp/~shinpei/papers/pro13.pdf, 2013.

[102] Shinpei Kato, Jason Aumiller, and Scott Brandt. Zero-copy i/o processing for low-latency gpu computing. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, pages 170–178. ACM, 2013.

[103] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 57–66. IEEE, 2011.

[104] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, page 17, 2011.

[105] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, 2012.

[106] Shinpei Kato and Nobuyuki Yamasaki. Portioned edf-based scheduling on multiprocessors. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 139–148, New York, NY, USA, 2008. ACM.

[107] Khronos OpenCL Working Group. The OpenCL specification. Available online – www.khronos.org/registry/cl/specs/opencl-1.2.pdf#page=49, 2011.

[108] Junsung Kim, Bjorn Andersson, Dionisio de Niz, and Ragunathan Rajkumar. Segment-fixed priority scheduling for self-suspending real-time tasks. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 246–257. IEEE, 2013.

[109] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[110] Konstantinos Bletsas. Worst-case and Best-case Timing Analysis for Real-time Embedded Systems with Limited Parallelism. PhD thesis, 2007.

[111] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 469–478. IEEE, 2009.

[112] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 259–268. IEEE, 2010.

[113] Karthik Lakshmanan and Ragunathan Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 3–12. IEEE, 2010.

[114] M. R. Leadbetter, G. Lindgren, and H. Rootzén. *Extremes and Related Properties of Random Sequences and Processes*. Springer-Verlag, 1983.

[115] M.R. Leadbetter. Extremes and local dependence in stationary sequences. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 65(2), 1983.

[116] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *Computers, IEEE Transactions on*, 47(6):700–713, 1998.

[117] John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171. IEEE, 1989.

[118] John P Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, volume 90, pages 201–209, 1990.

[119] John P Lehoczky. Real-time queueing theory. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 186–195. IEEE, 1996.

[120] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[121] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Analysis of global edf for parallel tasks. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 3–13. IEEE, 2013.

[122] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Christopher Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 85–96. IEEE, 2014.

[123] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration, 1995.

[124] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, (2):39–55, 2008.

[125] Björn Lisper. Towards parallel programming models for predictability. In Tullio Vardanega, editor, *Proc. 12th International Workshop on Worst-Case Execution-Time Analysis (WCET'12)*. Schloss Dagstuhl, July 2012.

[126] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for the Computing Machinery*, 20:46–61, 1973.

[127] Cong Liu and James Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 271–281. IEEE, 2013.

[128] Cong Liu and James Anderson. Suspension-Aware Analysis for Hard Real-Time Multiprocessor Scheduling. Technical report, Available at http://www.cs.unc.edu/ anderson/papers/ecrts13e_erratum.pdf, 2015.

[129] Cong Liu and James H Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 425–436. IEEE, 2009.

[130] Cong Liu and James H Anderson. Improving the schedulability of sporadic self-suspending soft real-time multiprocessor task systems. In *The Sixteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 13–22. IEEE, 2010.

[131] Cong Liu and James H Anderson. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 23–32. IEEE, 2010.

[132] Cong Liu and James H Anderson. Supporting soft real-time dag-based systems on multiprocessors with no utilization loss. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 3–13. IEEE, 2010.

[133] Cong Liu and James H Anderson. A new technique for analyzing soft real-time self-suspending task systems. *ACM SIGBED Review*, 9(4):29–32, 2012.

[134] Cong Liu and James H Anderson. An o (m) analysis technique for supporting real-time self-suspending task systems. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 373–382. IEEE, 2012.

[135] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[136] Jane WS Liu, Kwei-Jay Lin, Wei Kuan Shih, Albert Chuang-shi Yu, Jen-Yao Chung, and Wei Zhao. *Algorithms for scheduling imprecise computations*. Springer, 1991.

[137] R. M. Loynes. Extreme values in uniformly mixing stationary stochastic processes. *The Annals of Mathematical Statistics*, 36(3):993–999, 06 1965.

[138] Yue Lu, Thomas Nolte, Iain Bate, and Liliana Cucu-Grosjean. A new way about using statistical analysis of worst-case execution times. *ACM SIGBED Review*, 8(3):11–14, 2011.

[139] Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 12–21. IEEE, 1999.

[140] Irina Iulia Lupu and Joël Goossens. Scheduling of hard real-time multi-thread periodic tasks. In *Real-Time and Network Systems*, pages 35–44. Burns, Alan and George, Laurent, 2011.

[141] I. Majdandzic, C. Trefftz, and G. Wolffe. Computation of voronoi diagrams using a graphics processing unit. In *Proceedings of the IEEE International Conference on Electro/Information Technology (EIT 2008)*, pages 437 – 441, 2008.

[142] Rahul Mangharam and Aminreza Abrahimi Saba. Anytime algorithms for gpu architectures. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, December 2011.

[143] G Manimaran, C Siva Ram Murthy, and Krithi Ramamritham. A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. *Real-Time Systems*, 15(1):39–60, 1998.

[144] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 211–221. IEEE, 2015.

[145] Richard Membarth, Jan-Hugo Lupp, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Dynamic task-scheduling and resource management for gpu accelerators in medical imaging. In *Architecture of Computing Systems–ARCS 2012*, pages 147–159. Springer, 2012.

[146] Aloysius K Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[147] Sani R Nassif, Nikil Mehta, and Yu Cao. A resilience roadmap. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1011–1016. European Design and Automation Association, 2010.

[148] Geoffrey Nelissen, José Fonseca, Gurulingesh Raravi, and Vincent Nélis. Timing analysis of fixed priority self-suspending sporadic tasks. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 80–89. IEEE, 2015.

[149] Anh Nguyen, Yusuke Fujii, Yuki Iida, Takuya Azumi, Nobuhiko Nishio, and Shinpei Kato. Reducing data copies between gpus and nics. In *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2014 IEEE International Conference on*, pages 37–42. IEEE, 2014.

[150] Hossein Tehrani Niknejad, Taiki Kawano, Mikio Shimizu, and Seiichi Mita. Vehicle detection using discriminatively trained part templates with variable size. In *Intelligent Vehicles Symposium (IV), 2012 IEEE*, pages 766–771. IEEE, 2012.

[151] PJ Northrop. Semiparametric estimation of the extremal index using block maxima. Technical report, Dept of Statistical Science, UCL, 2005.

[152] NVIDIA Corp. NVIDIA's next generation CUDA compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[153] NVIDIA Corp. NVIDIA GeForce GTX 680. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf, 2012.

[154] NVIDIA Corp. NVIDIA's next generation CUDA compute architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[155] NVIDIA Corp. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/index.html, 2014.

[156] NVIDIA Corp. Cuda parallel computing platform. Available online – http://www.nvidia.com/object/cuda_home_new.html, 2015.

[157] NVIDIA Corporation. GeForce 256 the world's first GPU. Product brief – http://www.nvidia.com/page/geforce256.html, 1999.

[158] NVIDIA Corporation. NVIDIA CUDA C programming guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/ C/doc/CUDA_C_Programming_Guide.pdf, 2012.

[159] D.-I. Oh and T. P. Baker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real Time Systems Journal*, 15(2):183–192, 1998.

[160] Greger Ottosson and Mikael Sjodin. Worst-case execution time analysis for modern hardware architectures. In *In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97*, pages 47–55, 1997.

[161] John K Ousterhout. Scheduling techniques for concurrebt systems. In *ICDCS*, volume 82, pages 22–30, 1982.

[162] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, pages 80–113, 2007.

[163] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2):115–128, 2000.

[164] IC Pyle and V Illingworth. The oxford dictionary of computing, 1997.

[165] Manar Qamhieh, Laurent George, and Serge Midonnet. A stretching algorithm for parallel real-time dag tasks on multiprocessor systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, page 13. ACM, 2014.

[166] Brian Randell, Jean-Claude Laprie, Hermann Kopetz, and Bev Littlewood. *Predictably dependable computing systems*. Springer Science & Business Media, 2013.

[167] John Rice. *Mathematical statistics and data analysis*. Cengage Learning, 2006.

[168] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2008.

[169] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

[170] Dana Schaa and David Kaeli. Exploring the multiple-gpu design space. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, October 2009.

[171] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *Proceedings of the 16th Annual IEEE Symposium in Computer Science*, pages 151 – 162, 1975.

[172] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects.

[173] Jun Sun and Jane Liu. Synchronization protocols in distributed real-time systems. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 38–45. IEEE, 1996.

[174] Ola Svensson. Hardness of precedence constrained scheduling on identical machines. *SIAM Journal on Computing*, 40(5):1258–1274, 2011.

[175] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, 2000.

[176] T-S Tia, Zhong Deng, Mallikarjun Shankar, M Storch, Jun Sun, L-C Wu, and Jane W-S Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Real-Time Technology and Applications Symposium, 1995. Proceedings*, pages 164–173. IEEE, 1995.

[177] SR Srinivasa Varadhan. Asymptotic probabilities and differential equations. *Communications on Pure and Applied Mathematics*, 19(3):261–286, 1966.

[178] John Von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[179] G. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques Deuxième Mémoire: Recherches sûr les parallélloédres primitives. *Journal fur die Reine und Angewandte Mathematik*, 134:198 – 287, 1908.

[180] Wikipedia. Harvard Mark I. http://http://en.wikipedia.org/wiki/ Harvard_Mark_I, April 2015.

[181] Wikipedia. Host system. http://en.wikipedia.org/wiki/Host_system, April 2015.

[182] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem – overview of methods and survey of tools. *ACM Trans. Embedded Computing Systems*, 7(3):1–53, 2008.

[183] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):966–978, 2009.

[184] L. A. Wolsey and G. L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1998.

[185] Stephen John Young. *Real time languages*. Horwood, 1982.

[186] Y. Zhang and J.D. Owens. A quantitative performance analysis model for GPU architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. IEEE, 2011.